SEHH2239
Data Structures

Lecture 9

Tree II

Learning Objectives:

- **To describe the Binary Search Trees (BST)**
- **To implement BST by using a linked chain**
- **To use BST for searching and sorting**
- **To analysis BST and realize the need of balanced BST**
- **To operate the AVL Tree**

Binary Search Tree

Key-value pair and Search Tree

- **Key-value pair (KVP) is a set of two data items:**
  - ***Key*, which is a unique identifier for some item of data, and**
  - ***Value*, which is either the *data that is identified* or a *pointer* to the location of that data**
  - **Examples**
    - **(0, Apple), (1, Orange), (2, Pineapple),….**
    - **(002, “Chan Tia”), (004, “U Pei Yi”), (007, “Lee Chi Gi”)….**
    - **(“atmosphere”, “環境”), (“believe”, “相信”) (“mood”, 心境，心情，情緒；精神狀態”), (“process”, “過程；步驟”) ….**

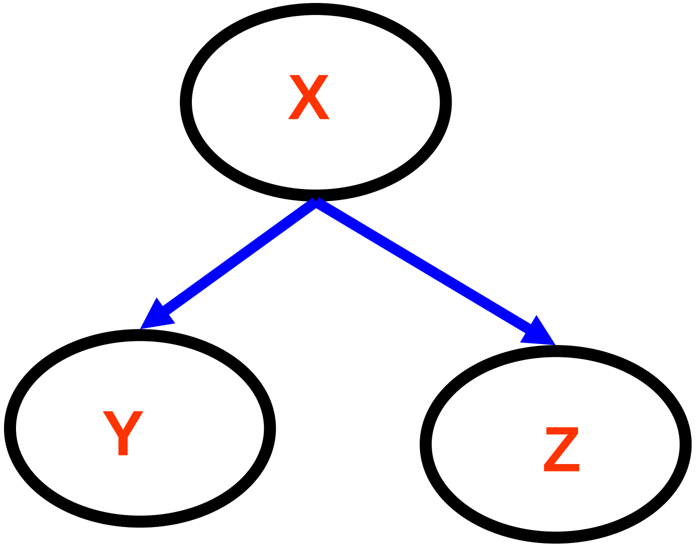
- **A Search Tree is a tree data structures that can be used to perform searching from a (key, value) pair.**

Definition Of Binary Search Tree

- A binary tree.
- Each node has a (key, value) pair.
- Nodes are insert to the tree according to the key.
- For every node x, all **keys** in the **left** subtree of x are **smaller** than the key in x.
- For every node x, all **keys** in the **right** subtree of x are **greater** than the key in x.
- No duplicate nodes.

Binary Search Trees

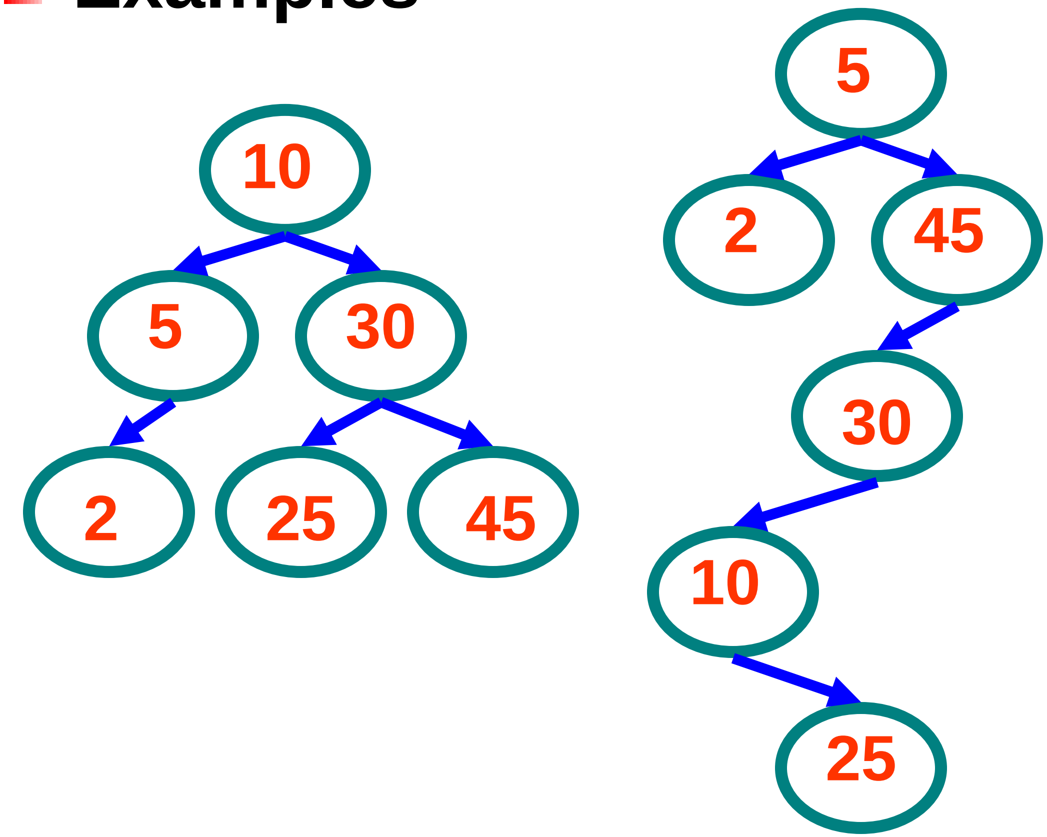
- **Key property**
  - **Value at node**
    - **Smaller values in left subtree**
    - **Larger values in right subtree**
  - **Example**
    - **X > Y**
    - **X < Z**



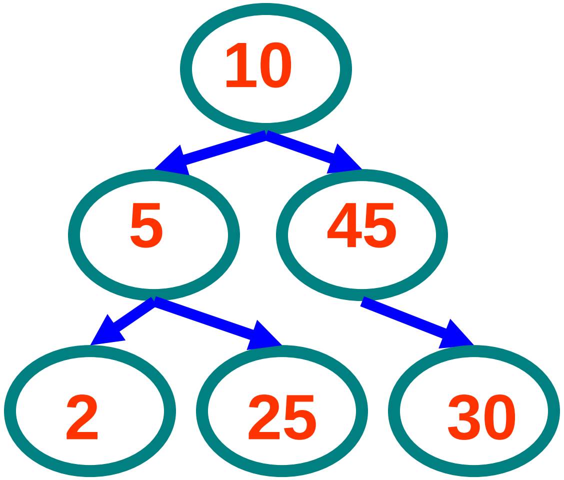
https://www.youtube.com/watch?v=mtvbVLK5xDQ
(00:00 – 02:00)

Binary Search Trees

- **Examples**

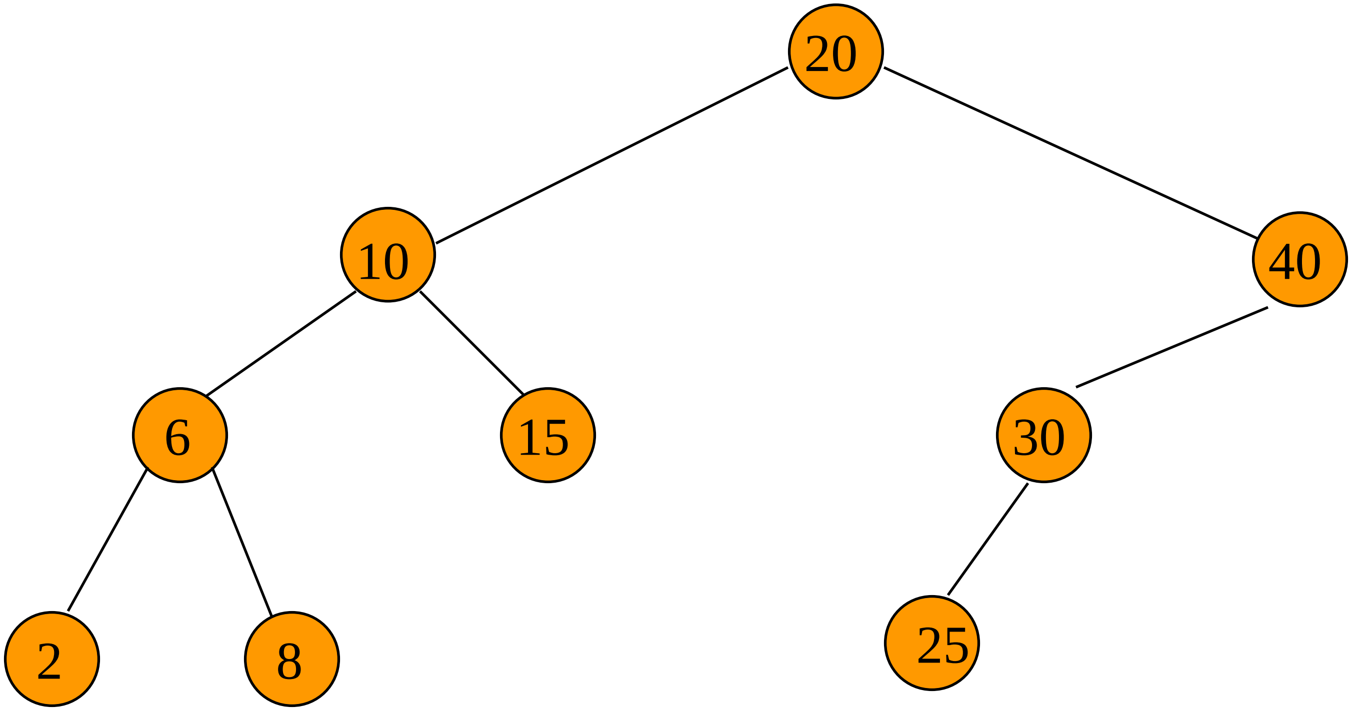


Binary search trees



Non-binary search tree

Example Binary Search Tree

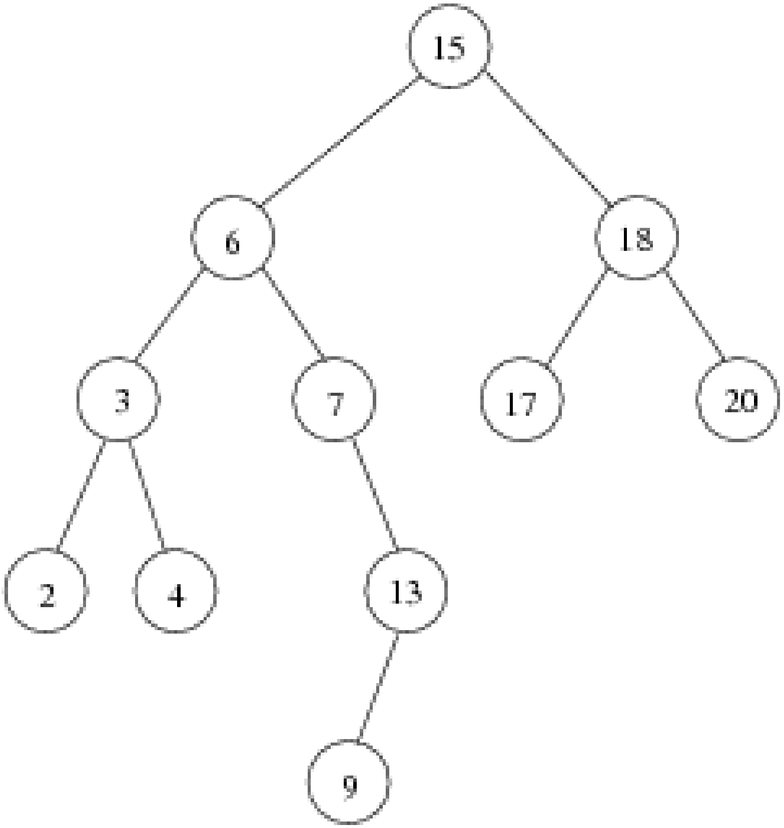


Insert 20, 10, 6, 2, 8, 15, 40, 30, 25

Only keys are shown.

Inorder traversal of BST

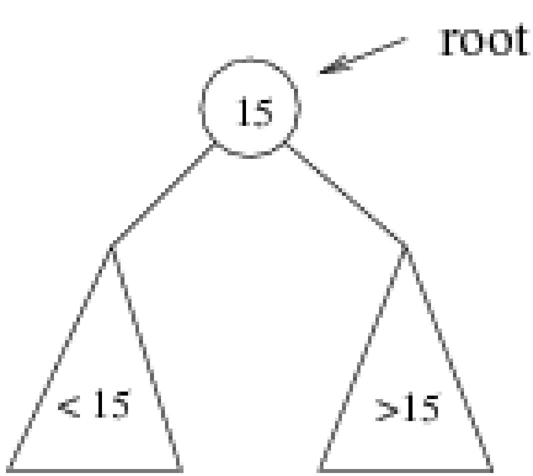
- **Print out all the keys in sorted order**



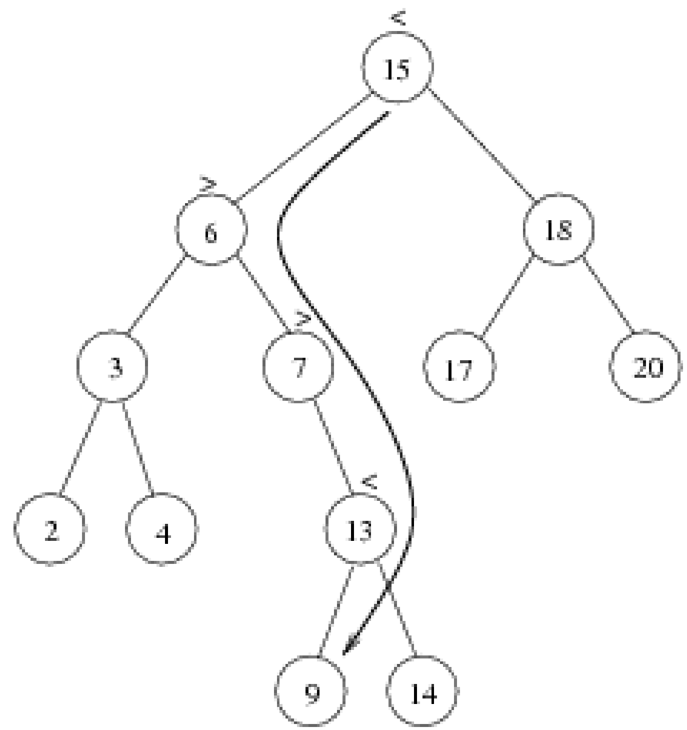
Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

Searching BST

- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.



Example: Search for 9 ...



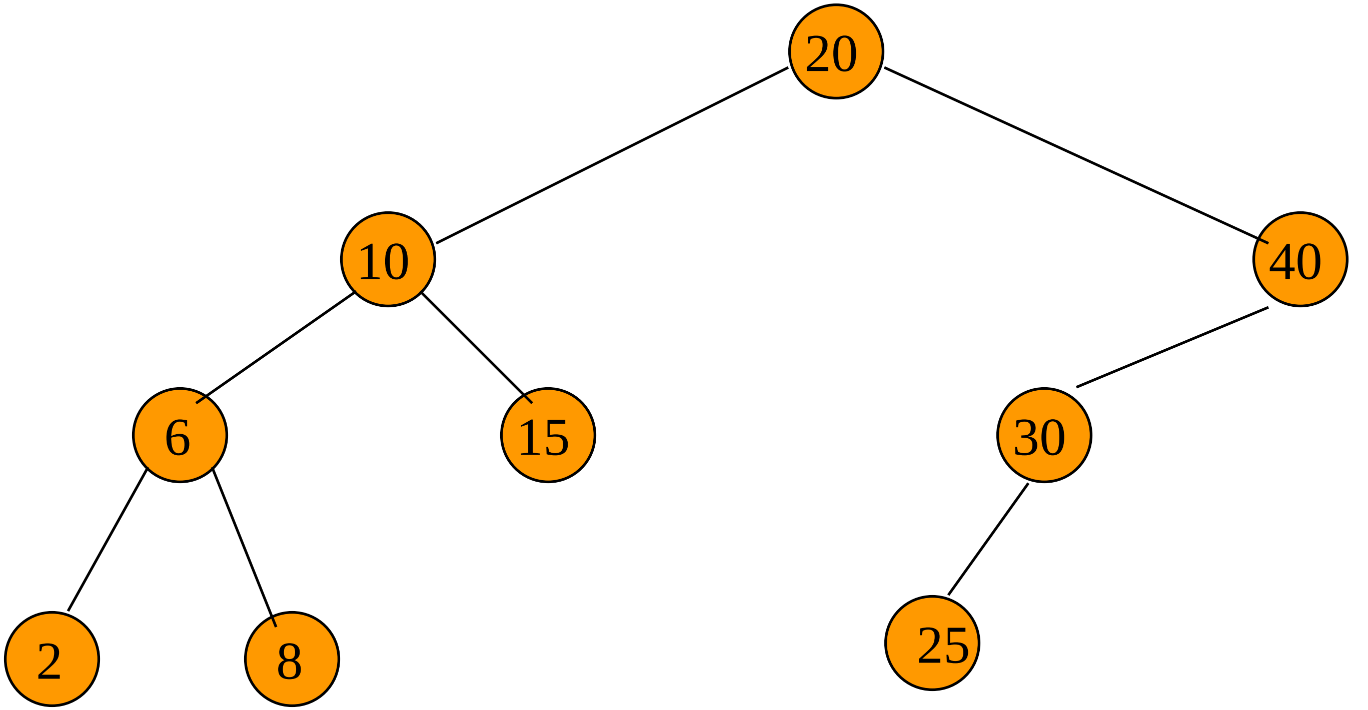
Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Key operations of BST

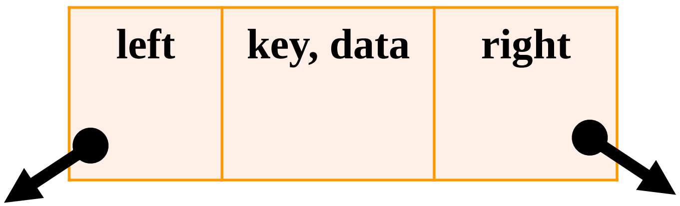
- get(k)
  - get the value *v* for a given key *k*
- put(k, v)
  - add a new (*k*, *v*) to a BST
- remove(k)
  - delete a node with key *k* from BST

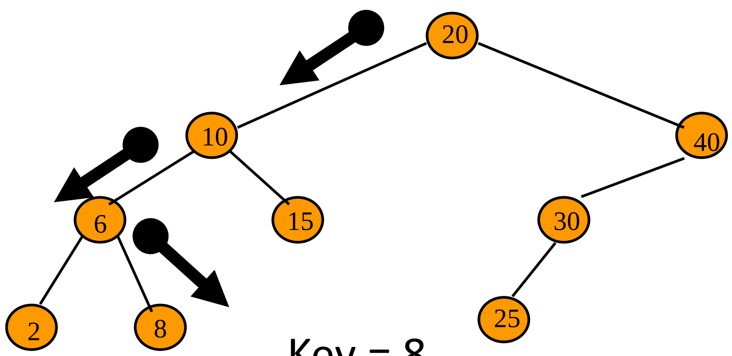
The Operation get()



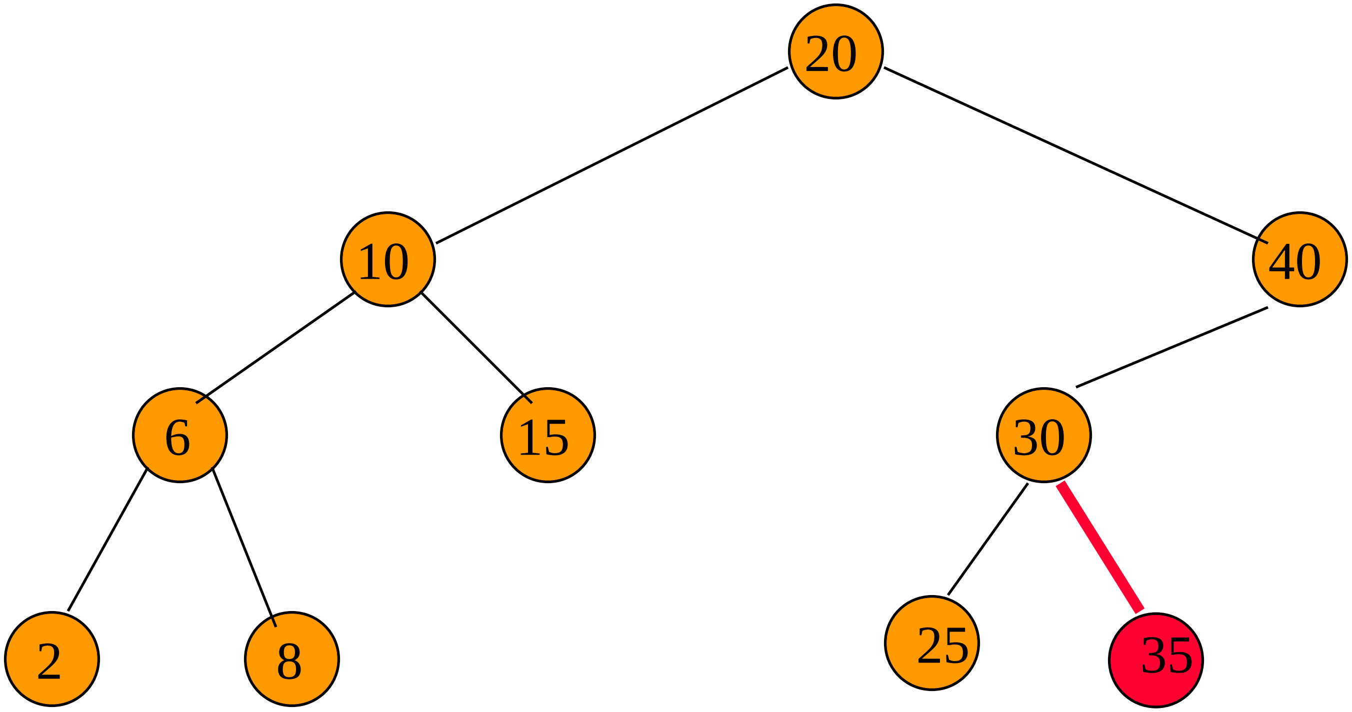
The Operation `get()`

```
class Node:
    def __init__(self, key, data):
        self.left = None
        self.right = None
        self.key = key
        self.data = data

    def get(self, key):
        p = self

        while p is not None:
          if p.key < key:
            p = p.left
          else:
            if p.key > key:
              p = p.right
            else:
              return p.data

        #no matching key
        return None
```



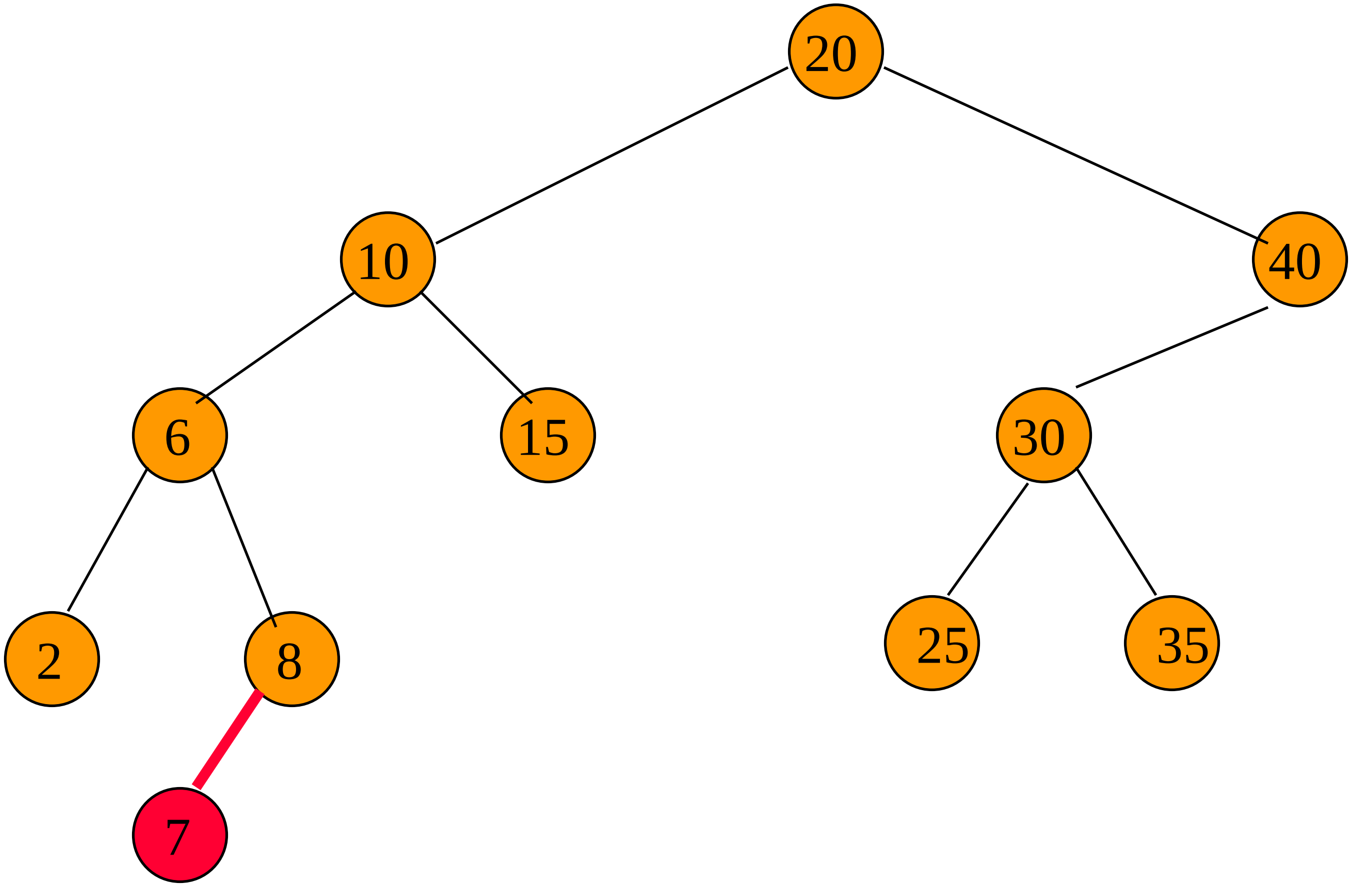


The Operation put()



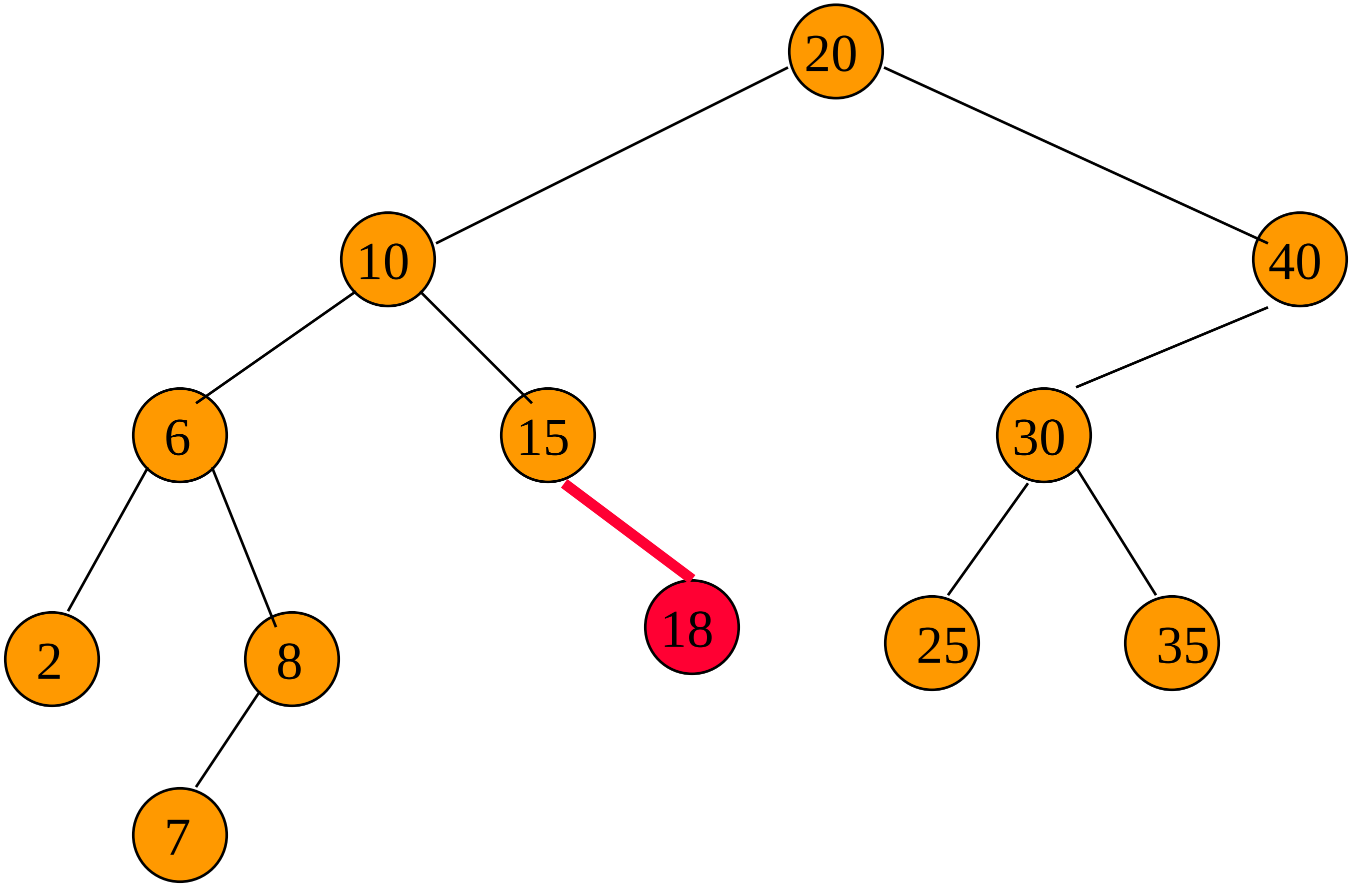
Put a pair whose key is 35.

The Operation put()



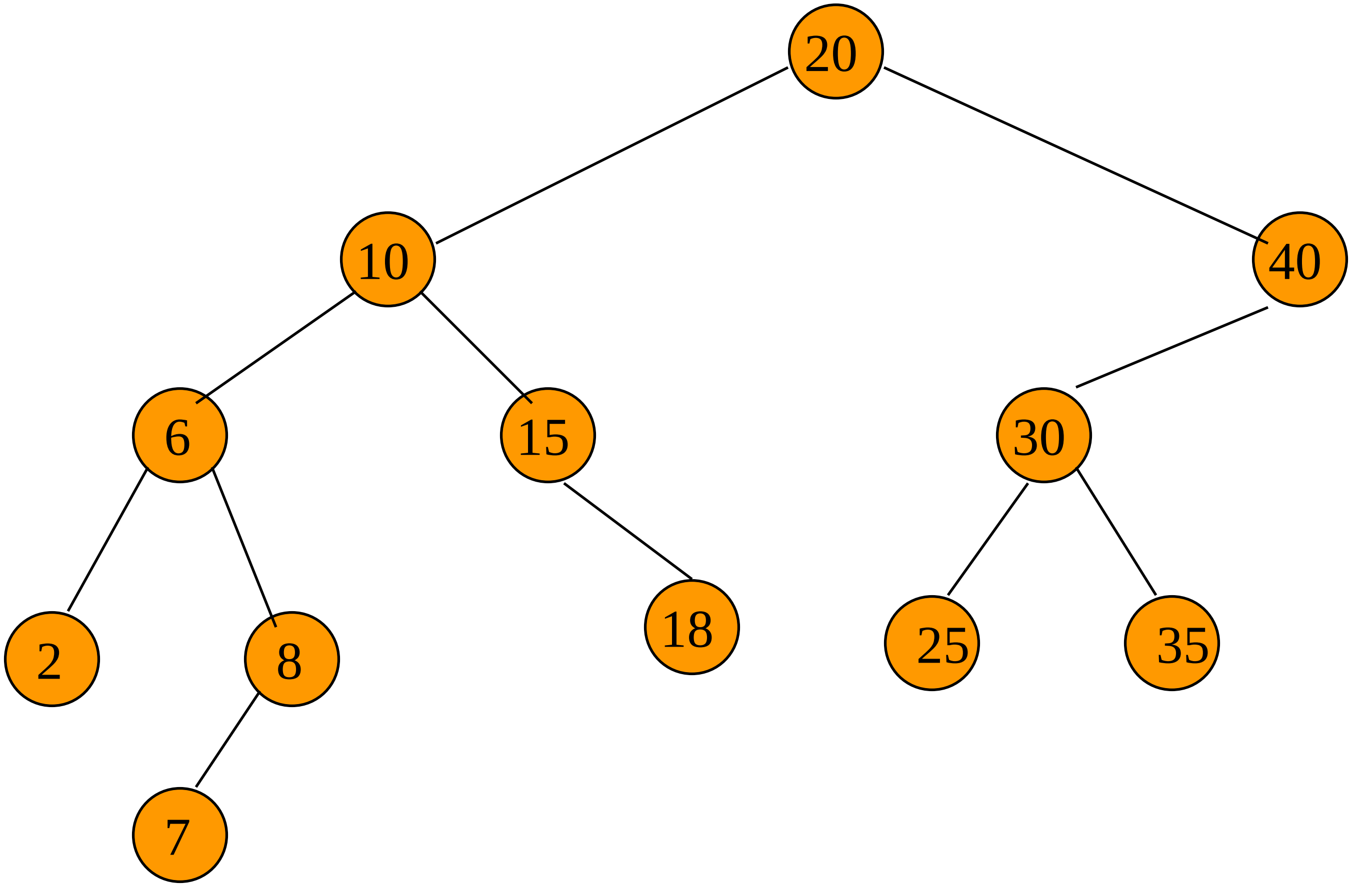
Put a pair whose key is 7.

The Operation put()



Put a pair whose key is 18.

The Operation put()



The Operation `put()`

```
def put(self, key, data):
     # Compare the new value with the parent node
    if self.key:
        if key < self.key:
            if self.left is None:
                self.left = Node(key, data)
            else:
                self.left.put(key, data)
        elif key > self.key:
                if self.right is None:
                    self.right = Node(key, data)
                else:
                    self.right.put(key, data)
    else:
        self.key = key
        self.data = data
```

Exercise

- For each of the following key sequences determine the binary search tree obtained when the keys are inserted one-by-one in the order given into an initially empty tree:

A. 1, 2, 3, 4, 5, 6, 7.

B. 4, 2, 1, 3, 6, 5, 7.

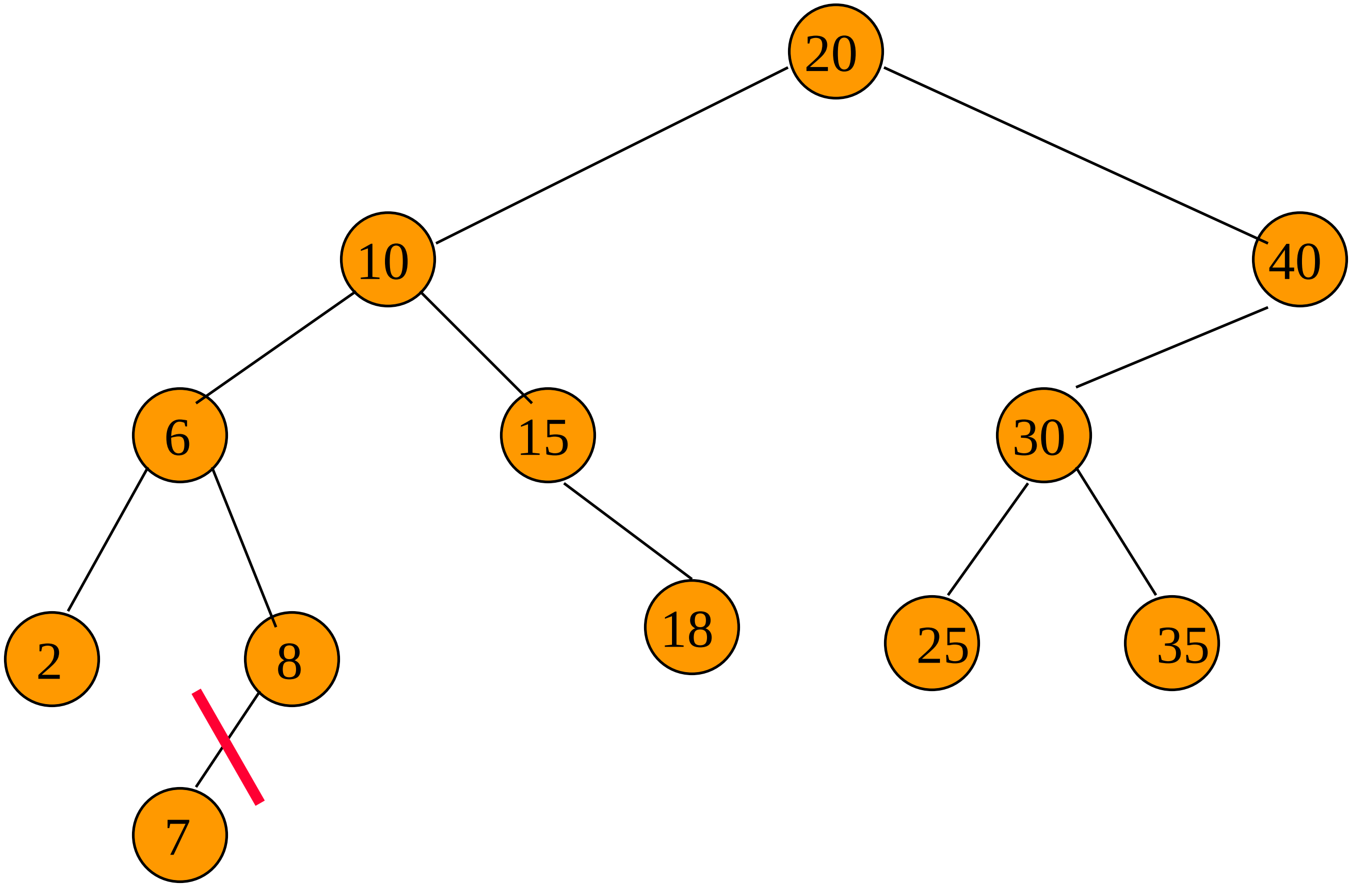
C. 1, 6, 7, 2, 4, 3, 5.

The Operation remove()

Three cases:

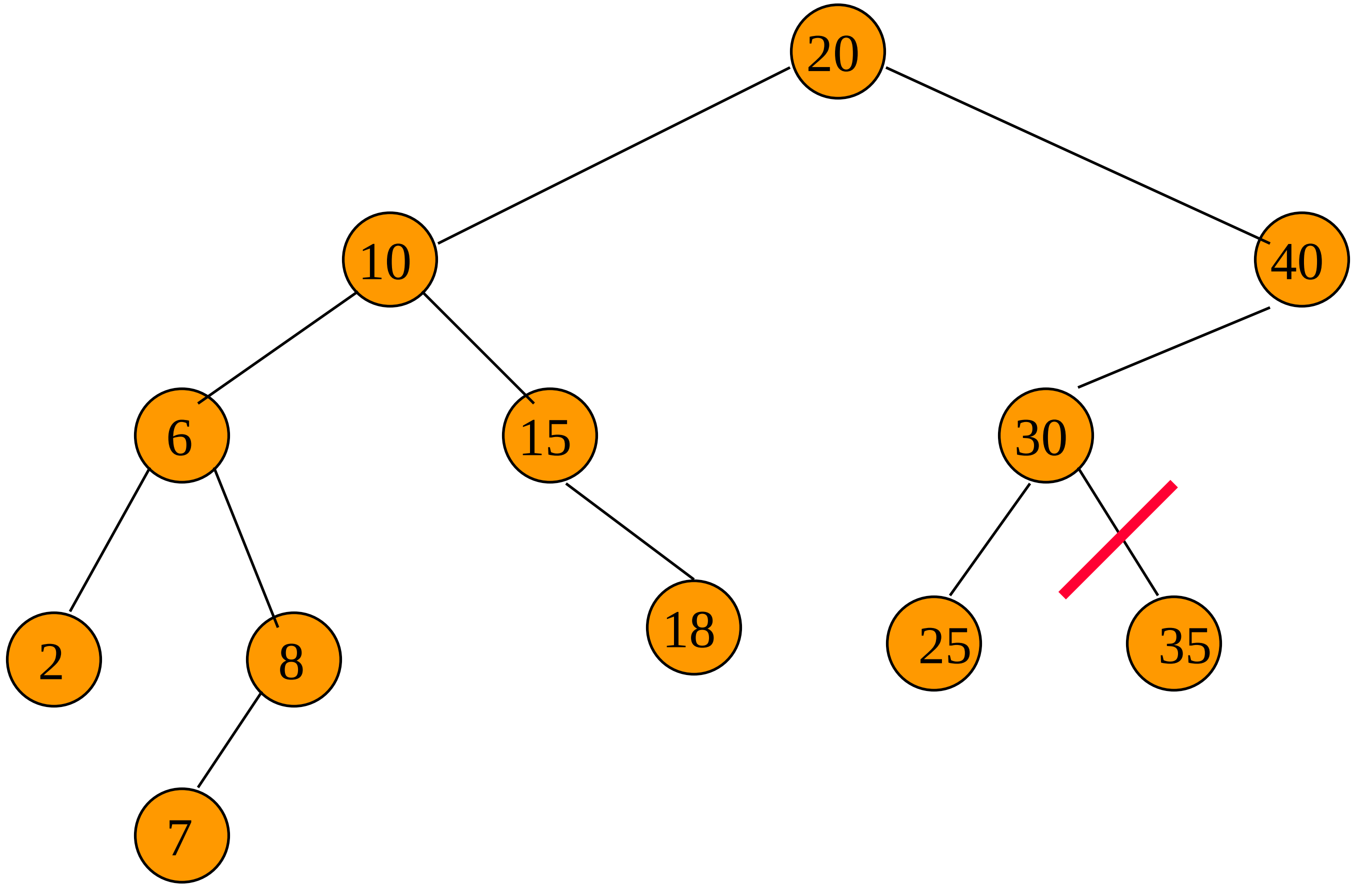
- Element is in a leaf.
- Element is in a degree 1 node.
- Element is in a degree 2 node.

Remove From A Leaf



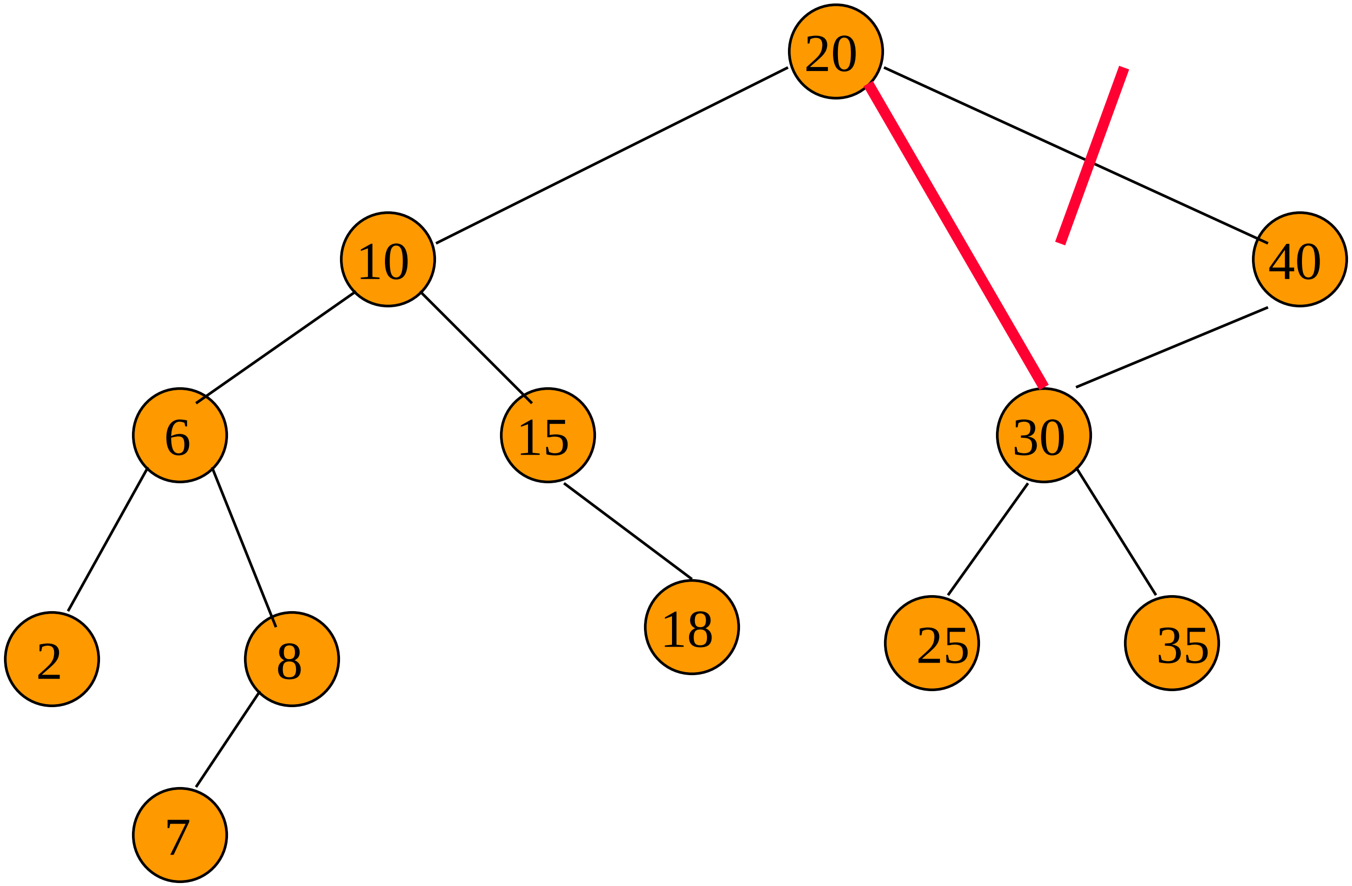
Remove a leaf element. key = 7

Remove From A Leaf (contd.)



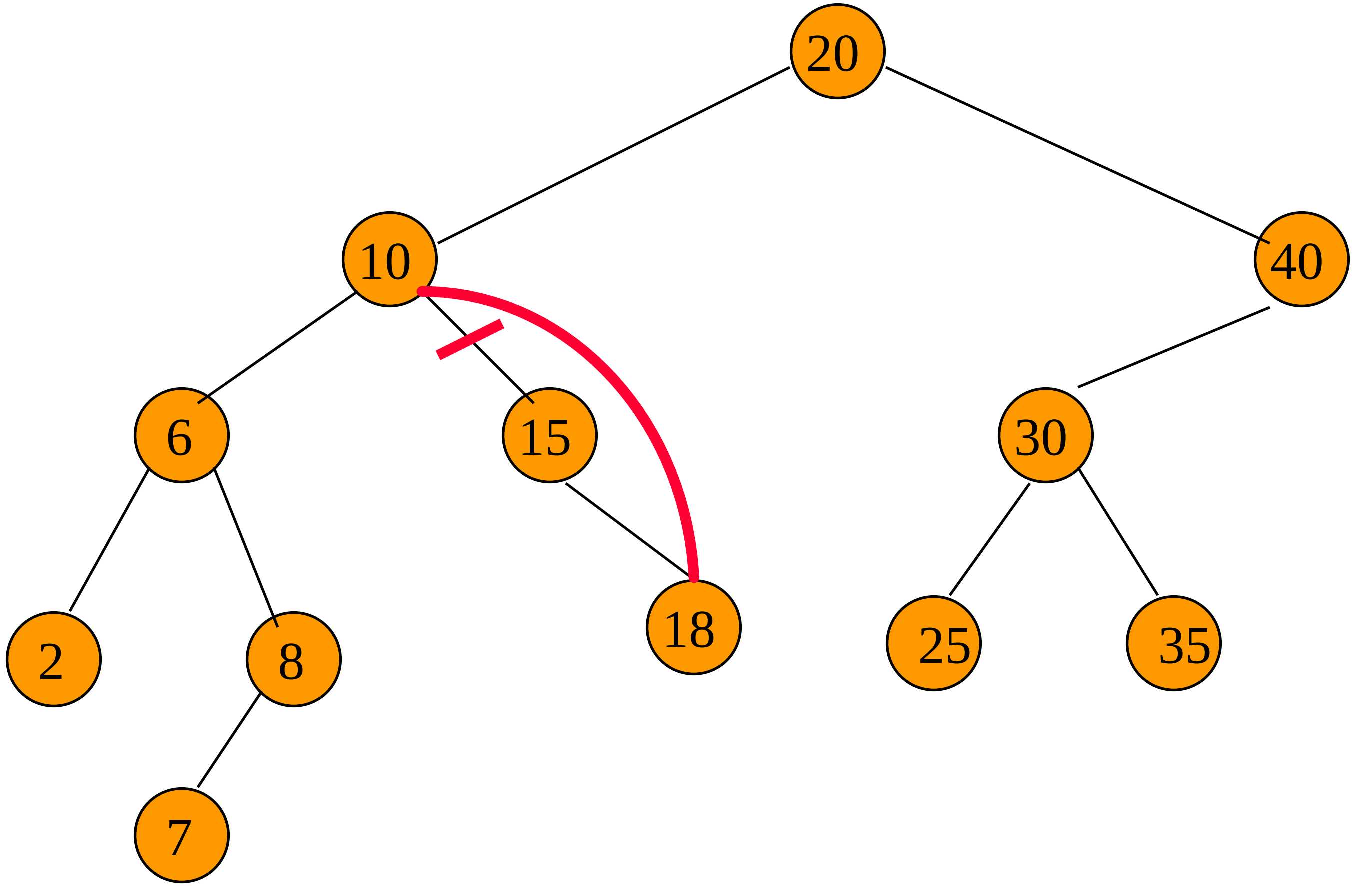
Remove a leaf element. key = 35

Remove From A Degree 1 Node



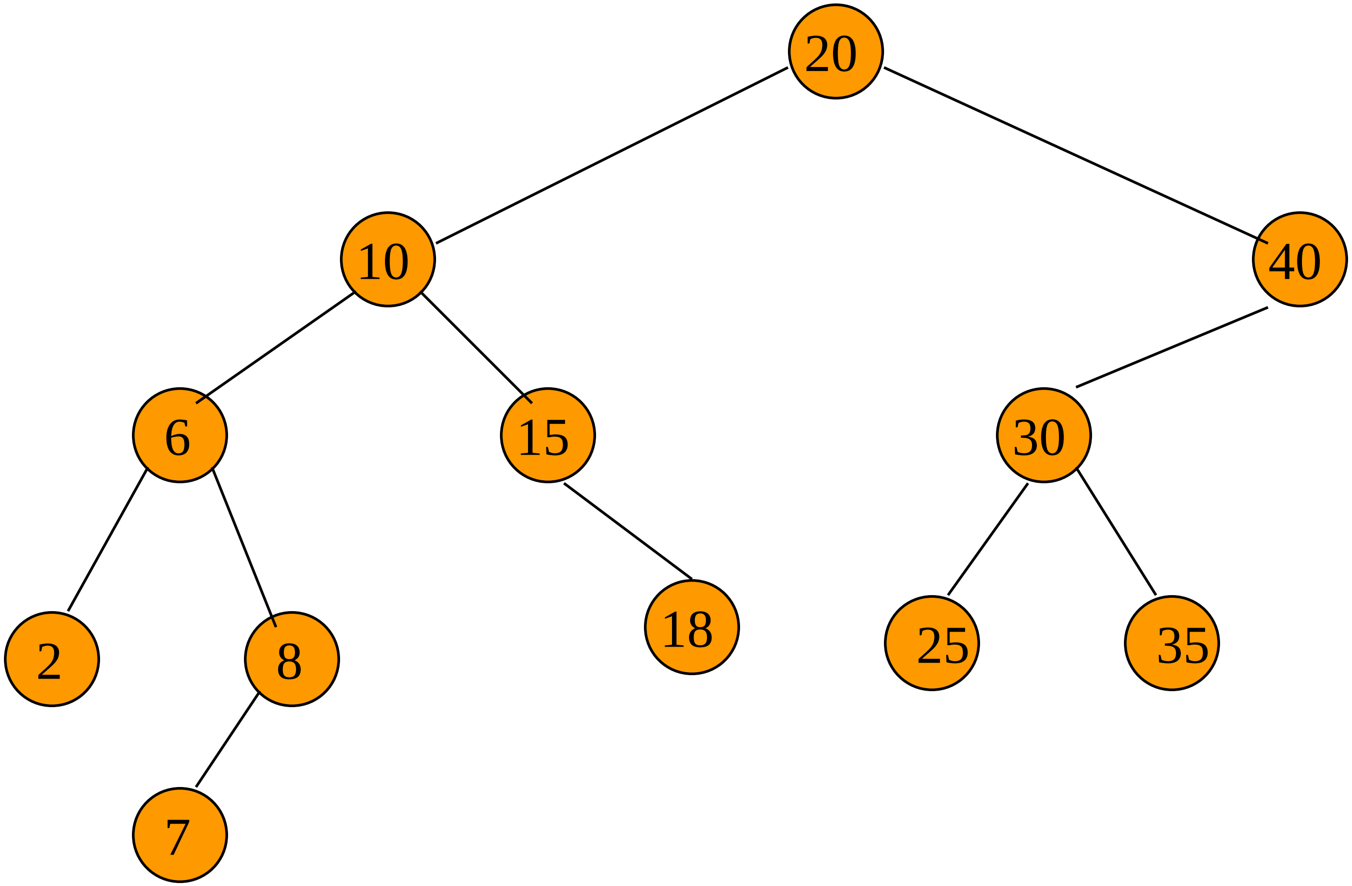
Remove from a degree 1 node. key = 40

Remove From A Degree 1 Node (contd.)



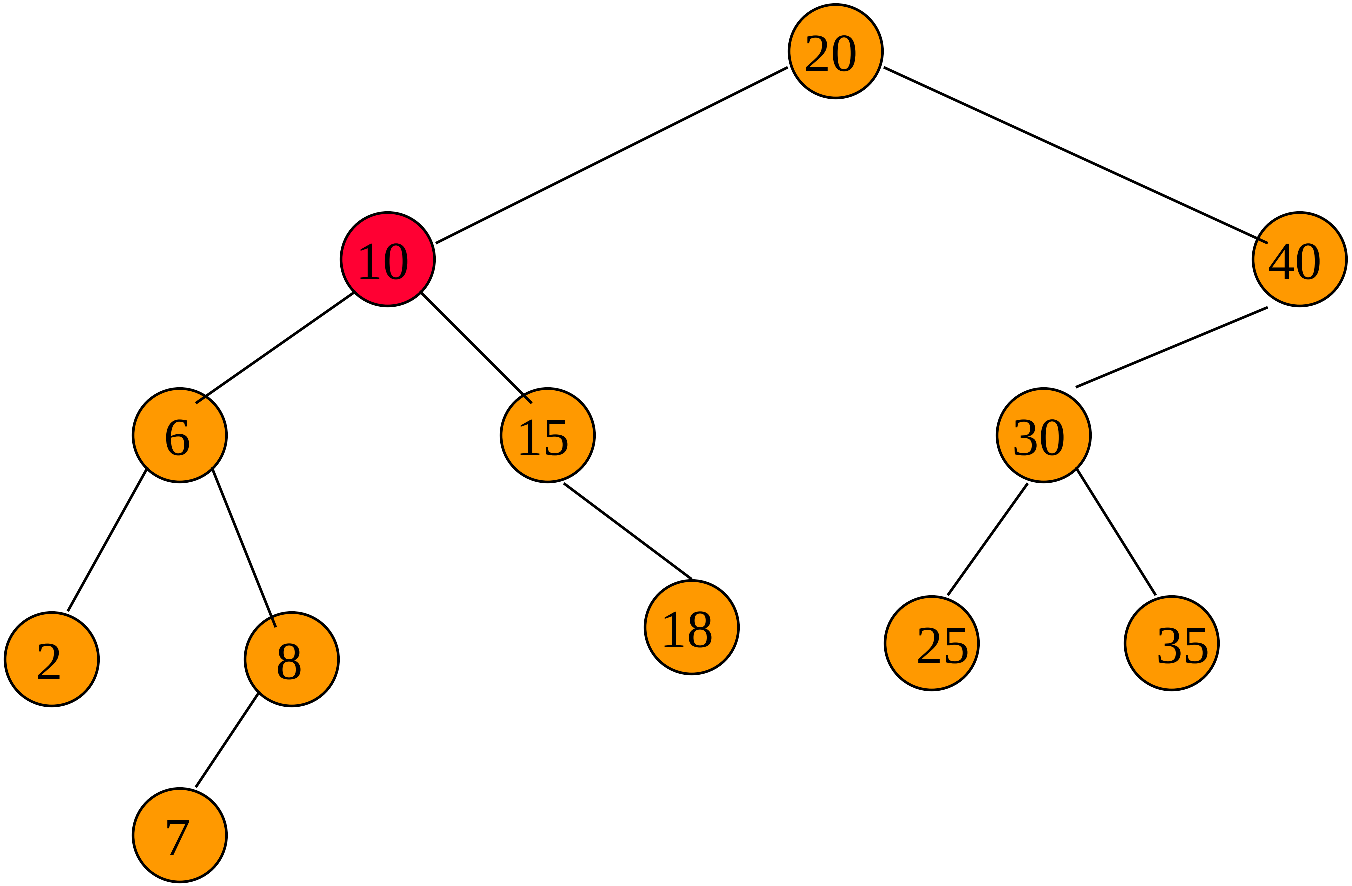
Remove from a degree 1 node. key = 15

Remove From A Degree 2 Node



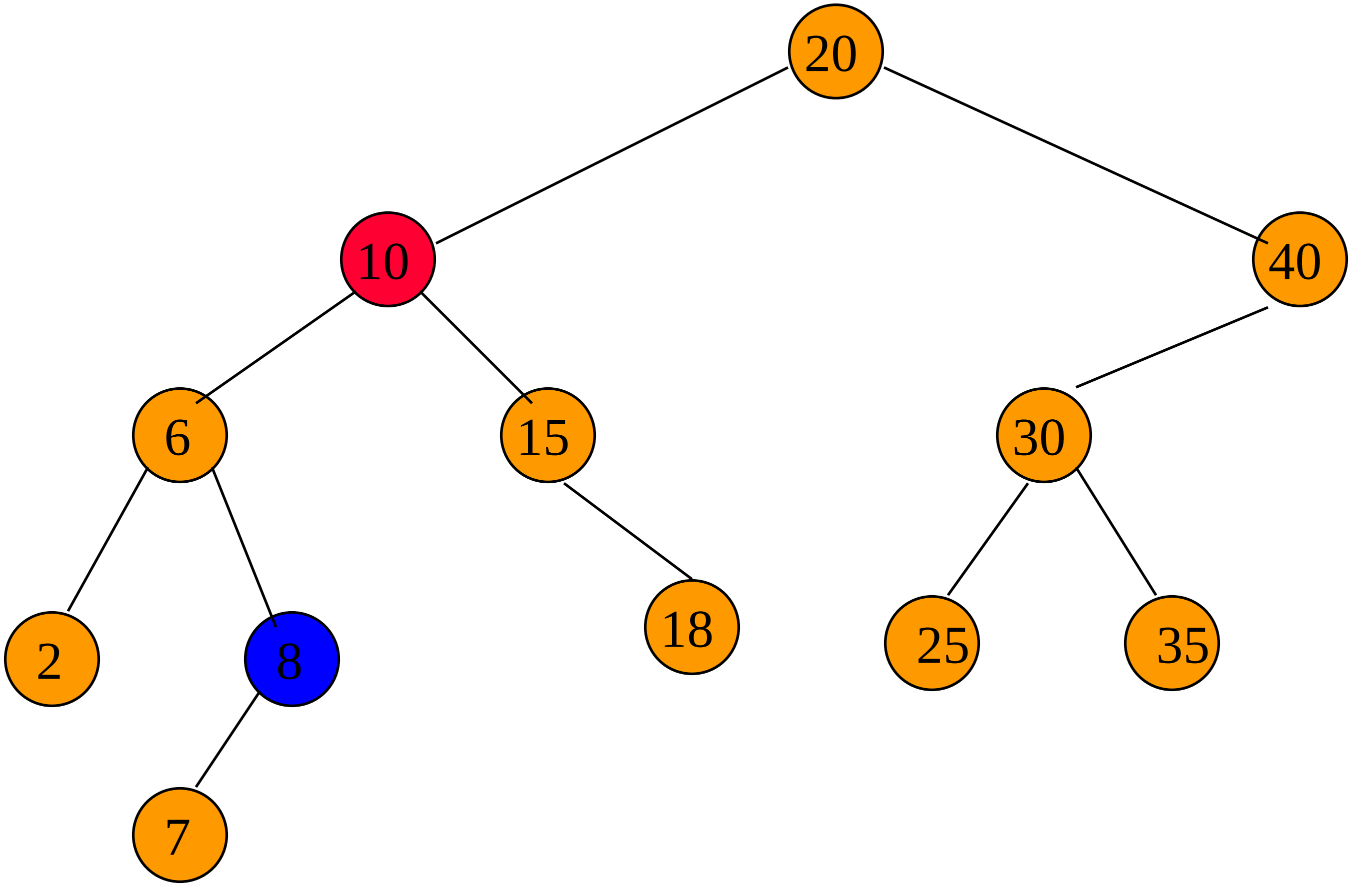
Remove from a degree 2 node. key = 10

Remove From A Degree 2 Node



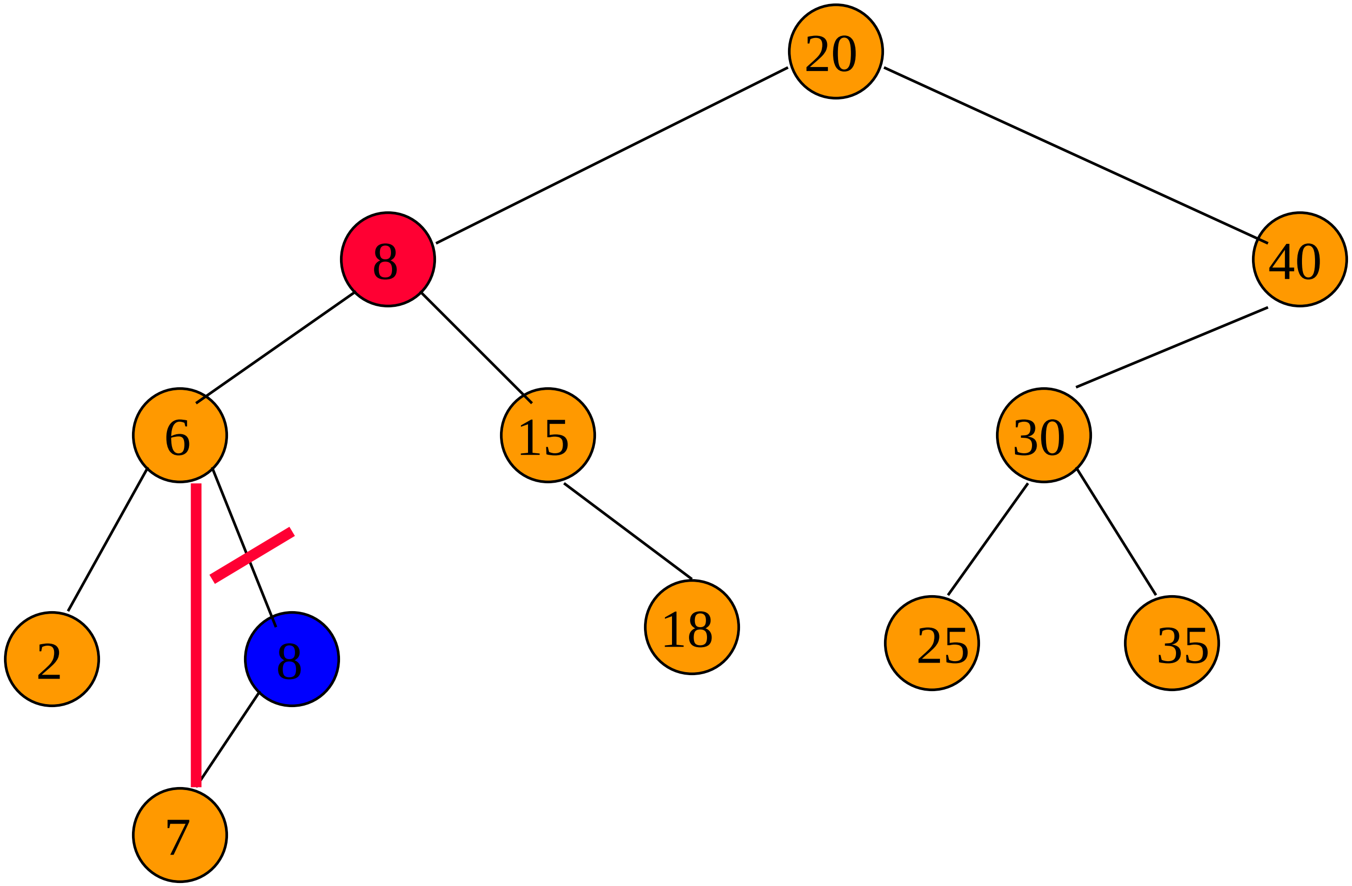
Replace with **largest** key in **left** subtree of the deleted node (or **smallest** in **right** subtree).

Remove From A Degree 2 Node



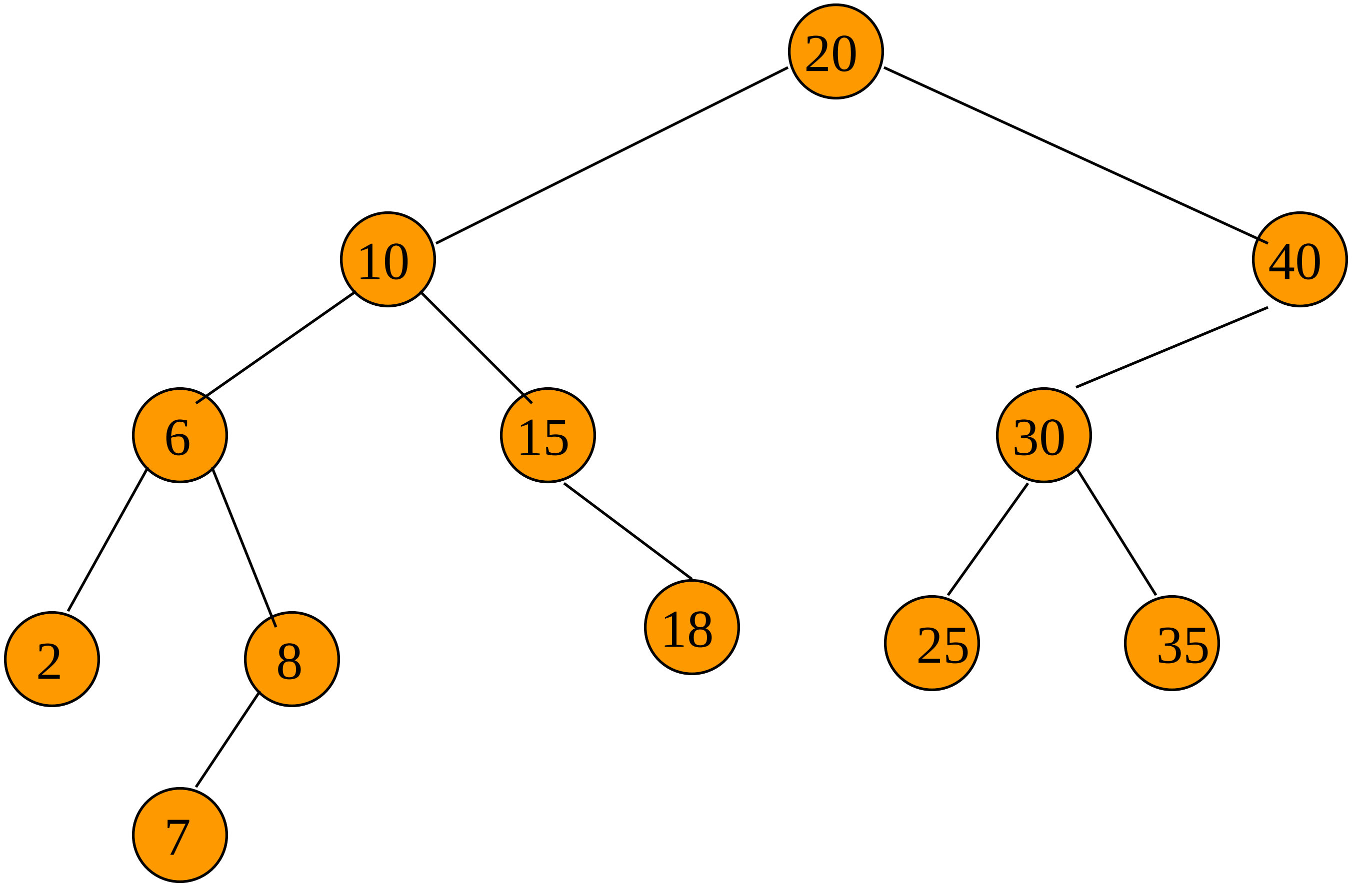
Replace with **largest** key in **left** subtree of the deleted node (or **smallest** in **right** subtree).

Remove From A Degree 2 Node



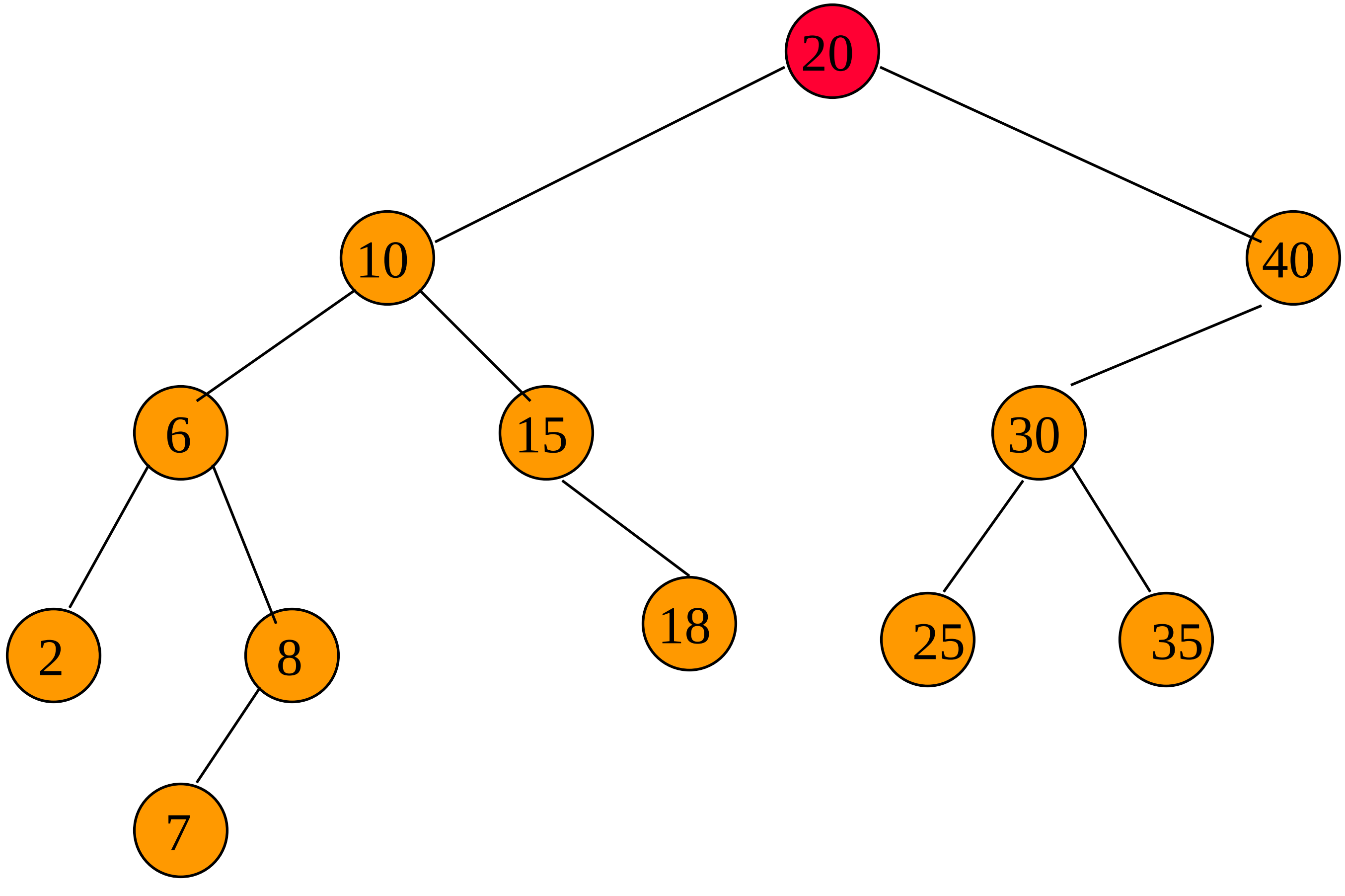
Largest key must be in a leaf or degree 1 node.

Another Remove From A Degree 2 Node



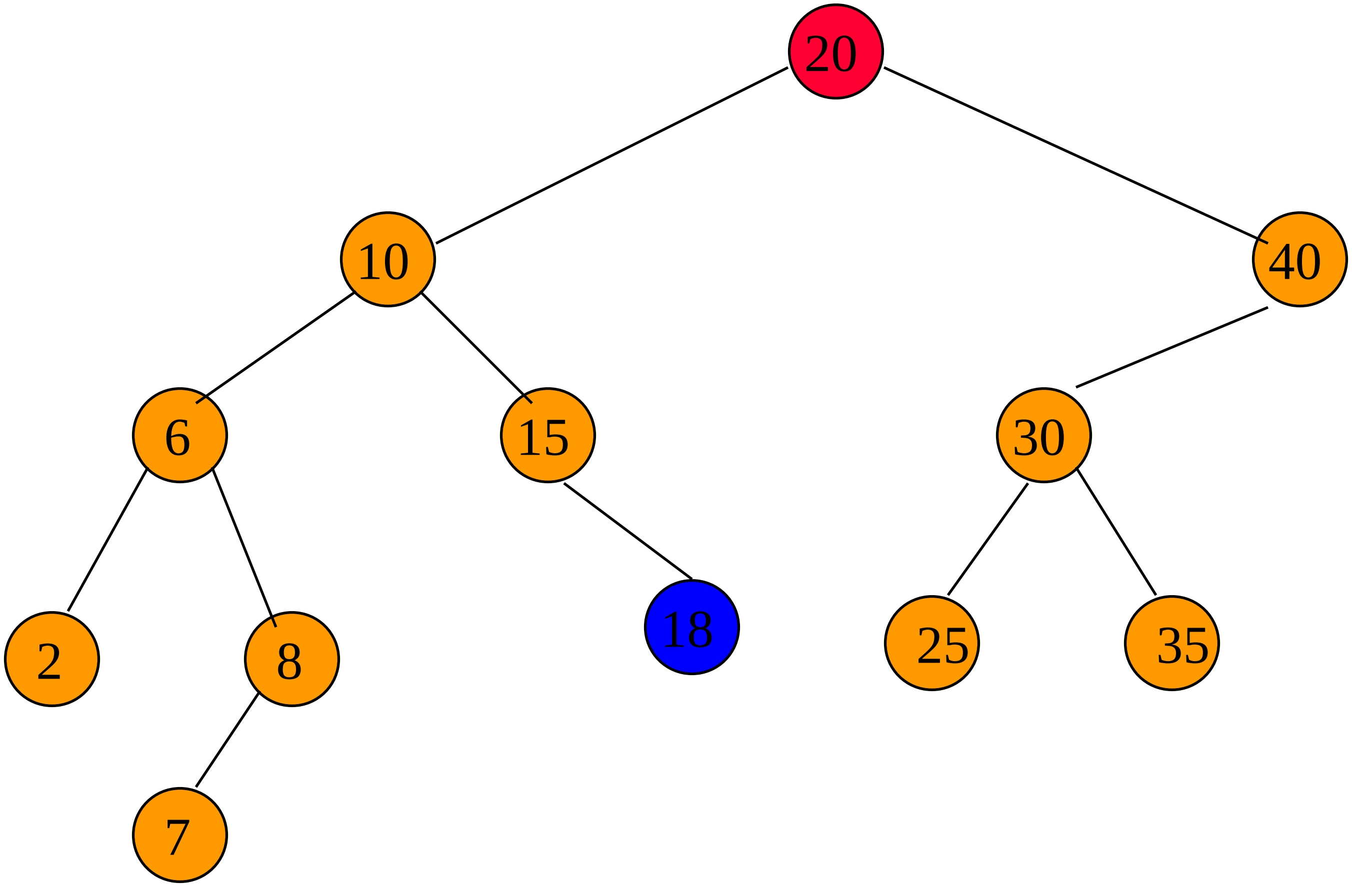
Remove from a degree 2 node. key = 20

Remove From A Degree 2 Node



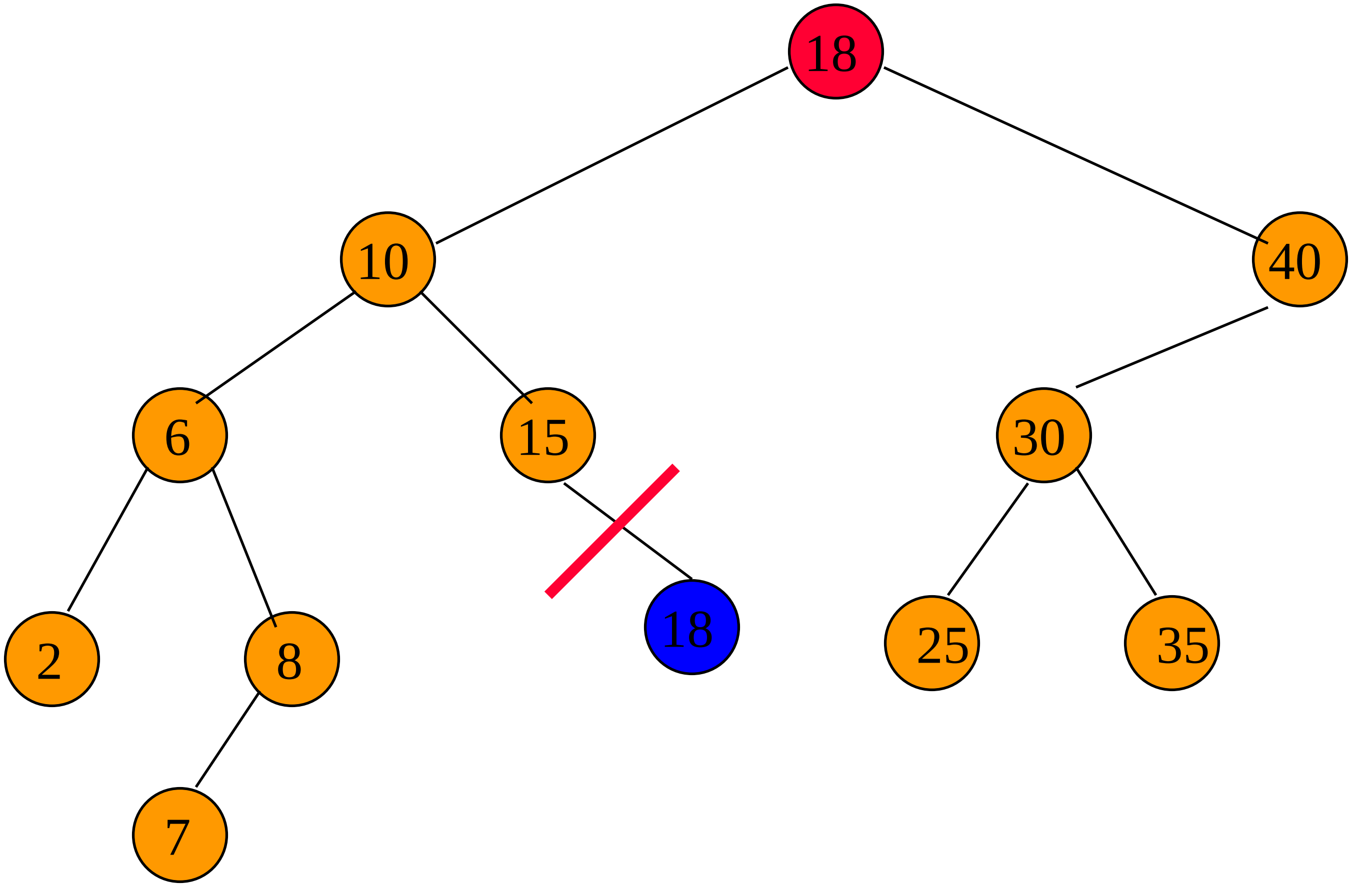
Replace with largest in left subtree of the deleted node.

Remove From A Degree 2 Node



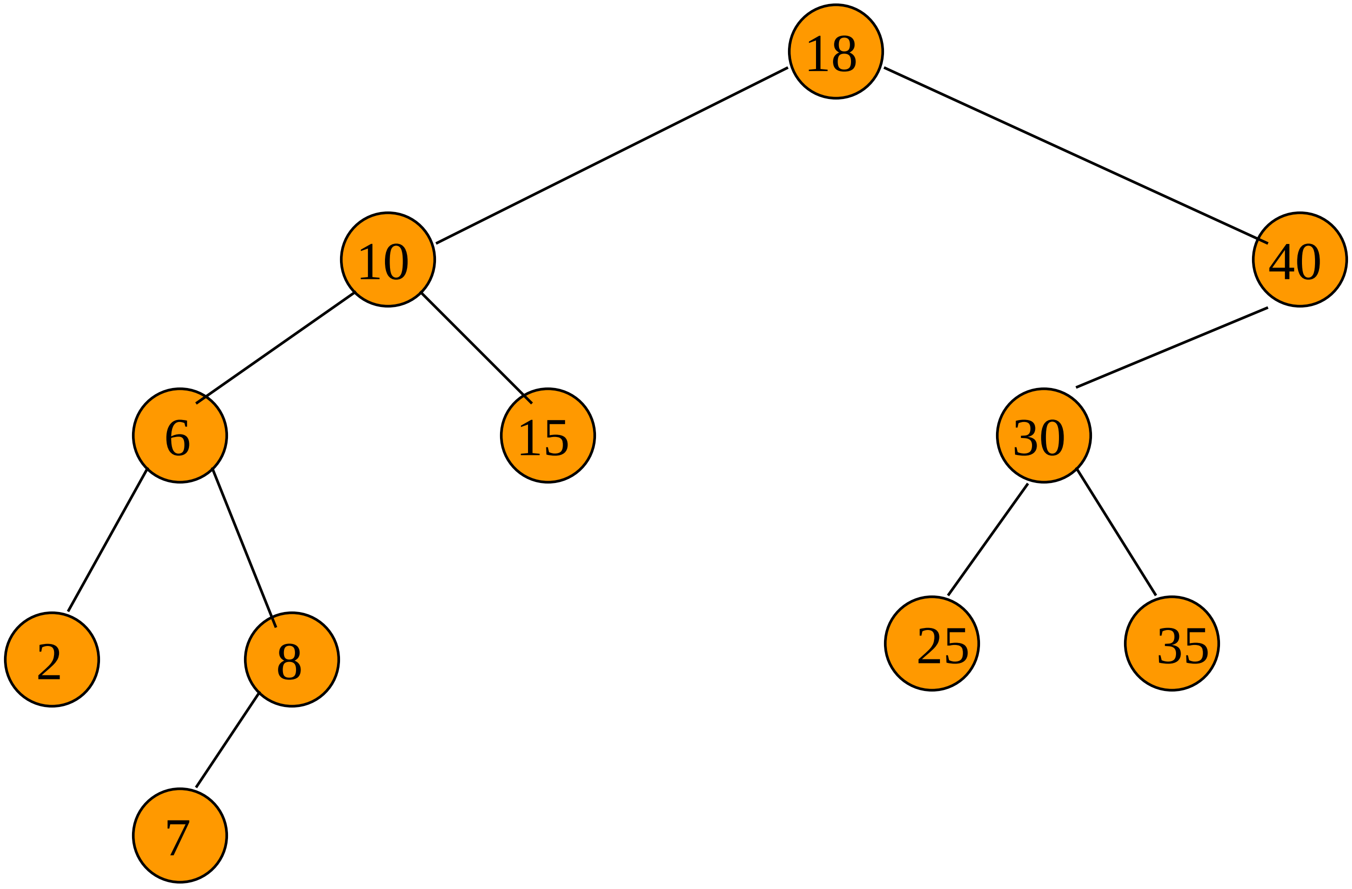
Replace with largest in left subtree of the deleted node.

Remove From A Degree 2 Node



Replace with largest in left subtree of the deleted node.

Remove From A Degree 2 Node



https://www.youtube.com/watch?v=mtvbVLK5xDQ
(2:00-6:00)

Find smallest

```
# To find the inorder successor which
# is the smallest node in the subtree

   def findsuccessor(self, node):
        current_node = node
        while current_node.left != None:
            current_node = current_node.left
        return current_node
```

Remove()

```
def remove(self, root, key):
    # Base Case
    if root is None:
        return root

    # If the key to be deleted
    # is smaller than the root's
    # key then it lies in left subtree
    if key < root.key:
        root.left = self.remove(root.left, key)

    # If the kye to be delete
    # is greater than the root's key
    # then it lies in right subtree
    elif(key > root.key):
        root.right = self.remove(root.right, key)
```

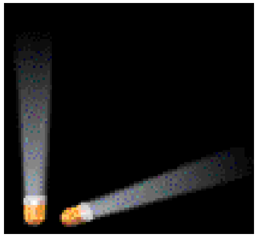
Remove()

```python
# If key is same as root's key, then this is the node
 # to be deleted
 else:

     # Node with only one child or no child
     if root.left is None:
         temp = root.right
         root = None
         return temp

     elif root.right is None:
         temp = root.left
         root = None
         return temp
```

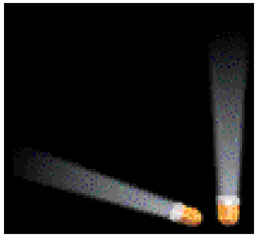
Remove()

```
    # Node with two children:
    # Get the inorder successor
    # (smallest in the right subtree)
    temp = self.findsuccessor(root.right)

    # Copy the inorder successor's
    # content to this node
    root.key = temp.key

    # Delete the inorder successor
    root.right = self.remove(root.right, temp.key)

return root
```

Balanced BST / AVL Tree



Balanced Binary Search Trees



- AVL (Adel'son-Vel'skii and Landis) trees
- Hopefully, height is $O(\log_2 n)$, where n is the number of elements in the tree
- Ideally, get, put, and remove take $O(\log_2 n)$ time

AVL Tree

- binary tree
- for every node x, define its balance factor

  balance factor of x = height of right subtree of x
  - height of left subtree of x

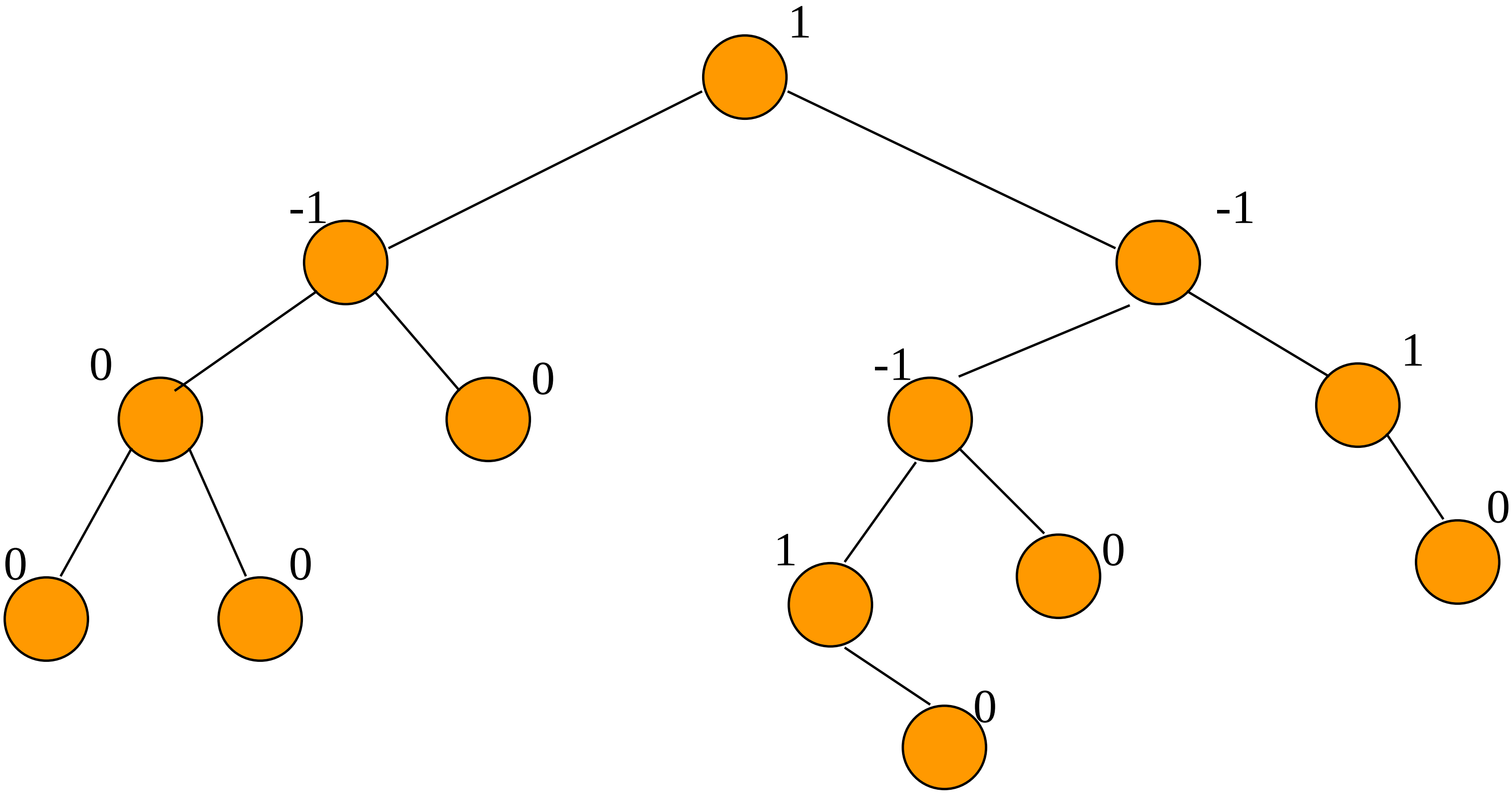
- balance factor of every node x is -1, 0, or 1

Note: In some texts (e.g. the textbook by Sahni), the balance factor is computed as follows:

balance factor of x = height of left subtree of x - height of right subtree of x

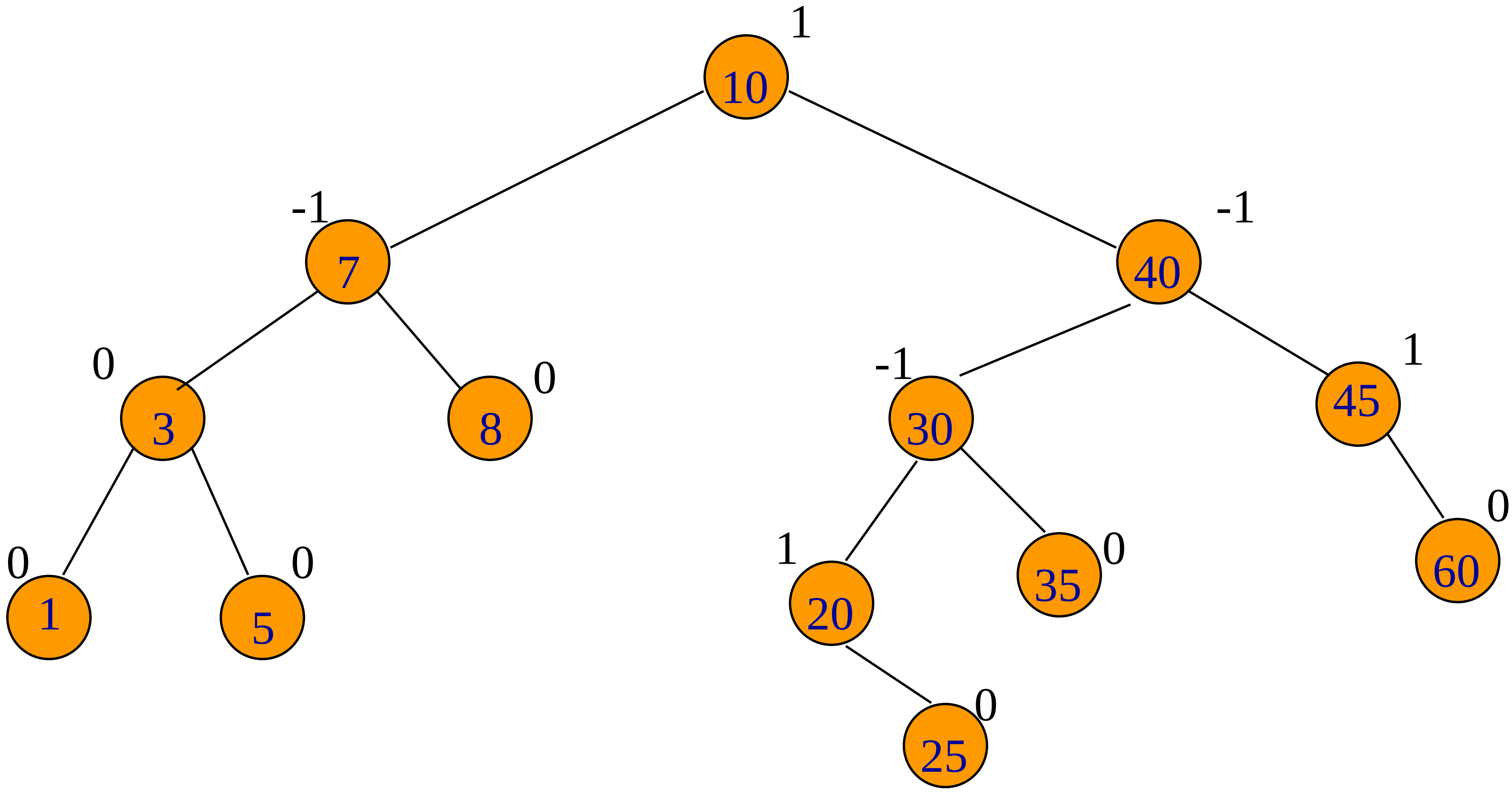
This note uses the above-mentioned implementation.

Balance Factors

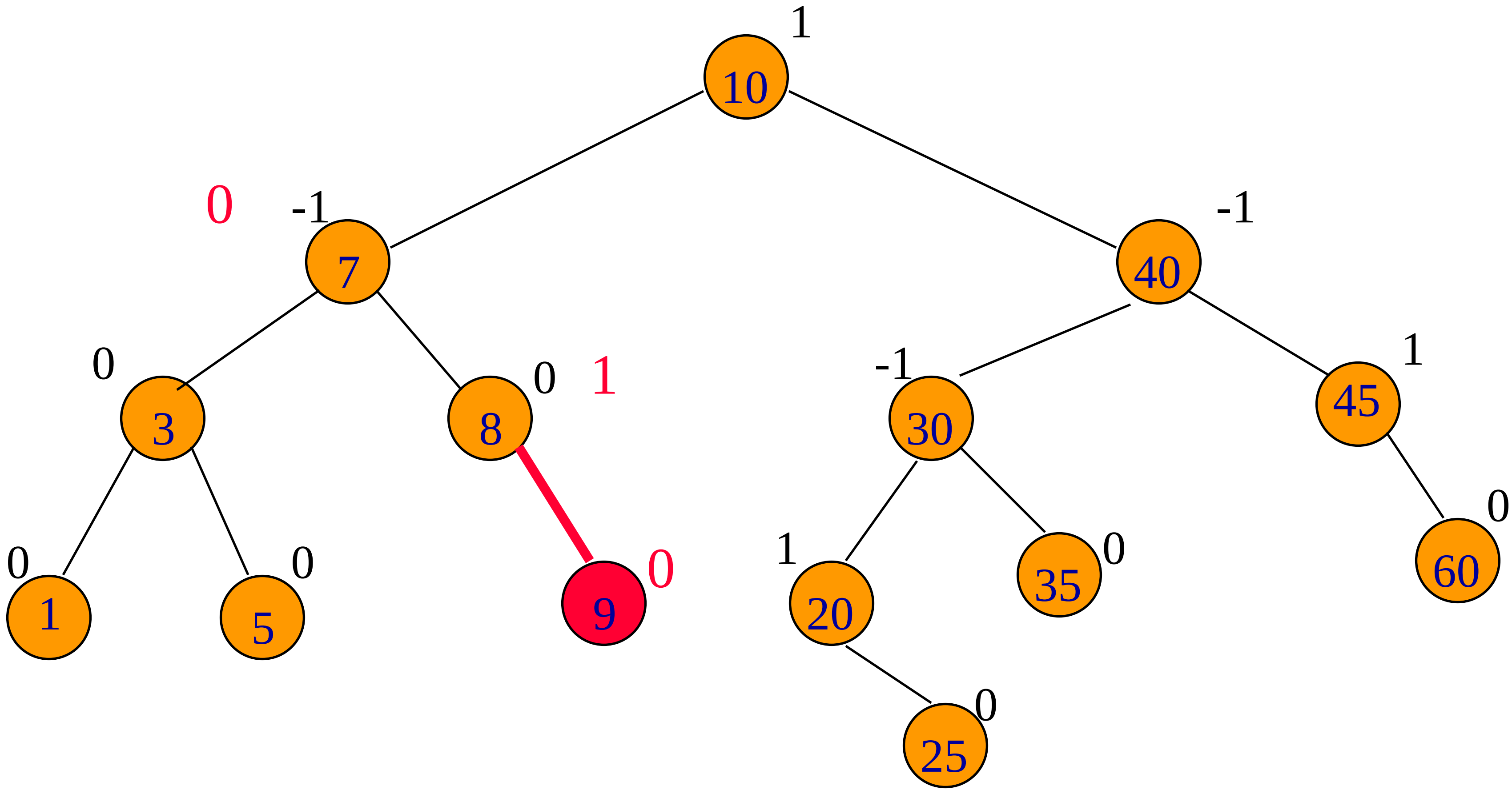


This is an AVL tree.

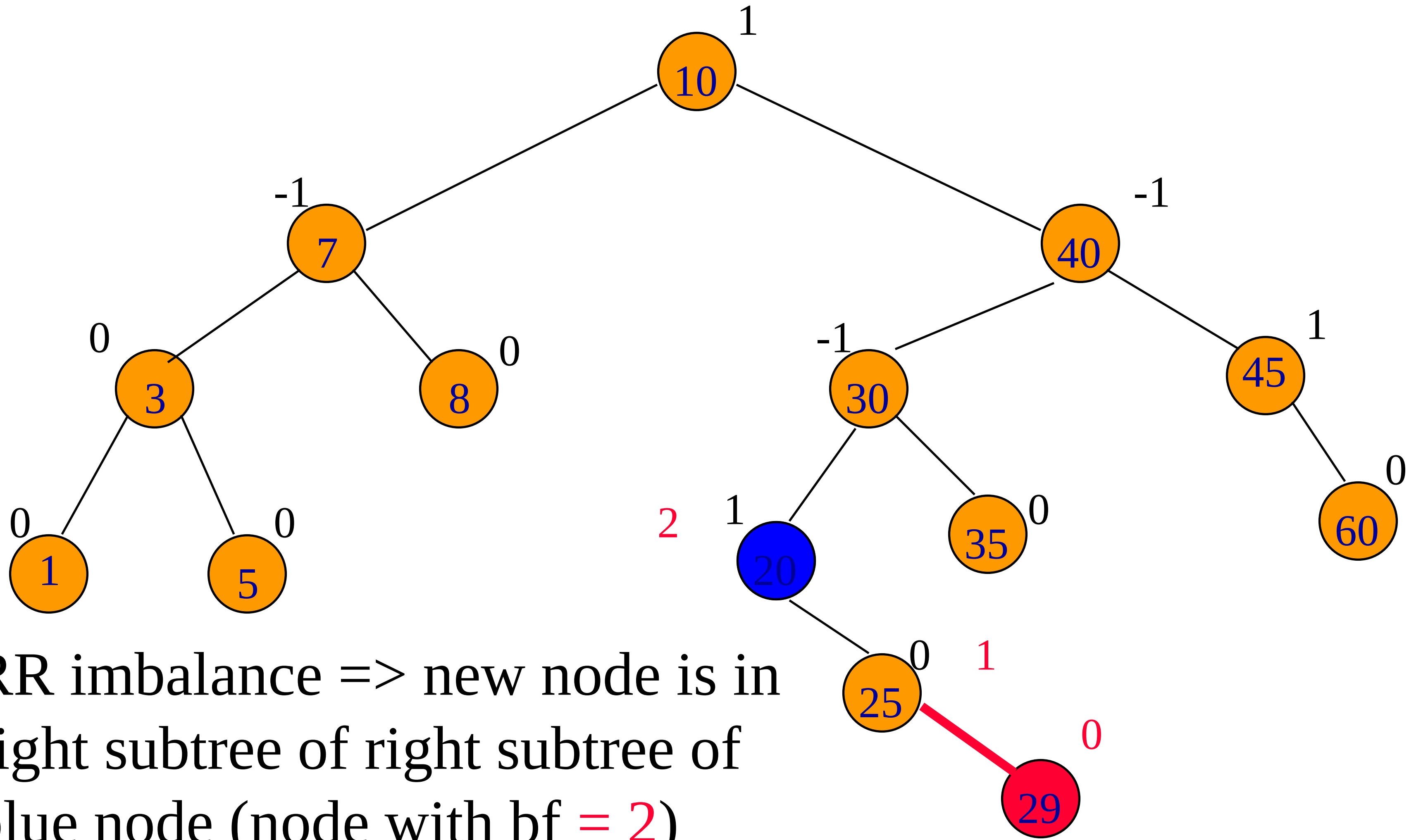
AVL Search Tree



put(9)

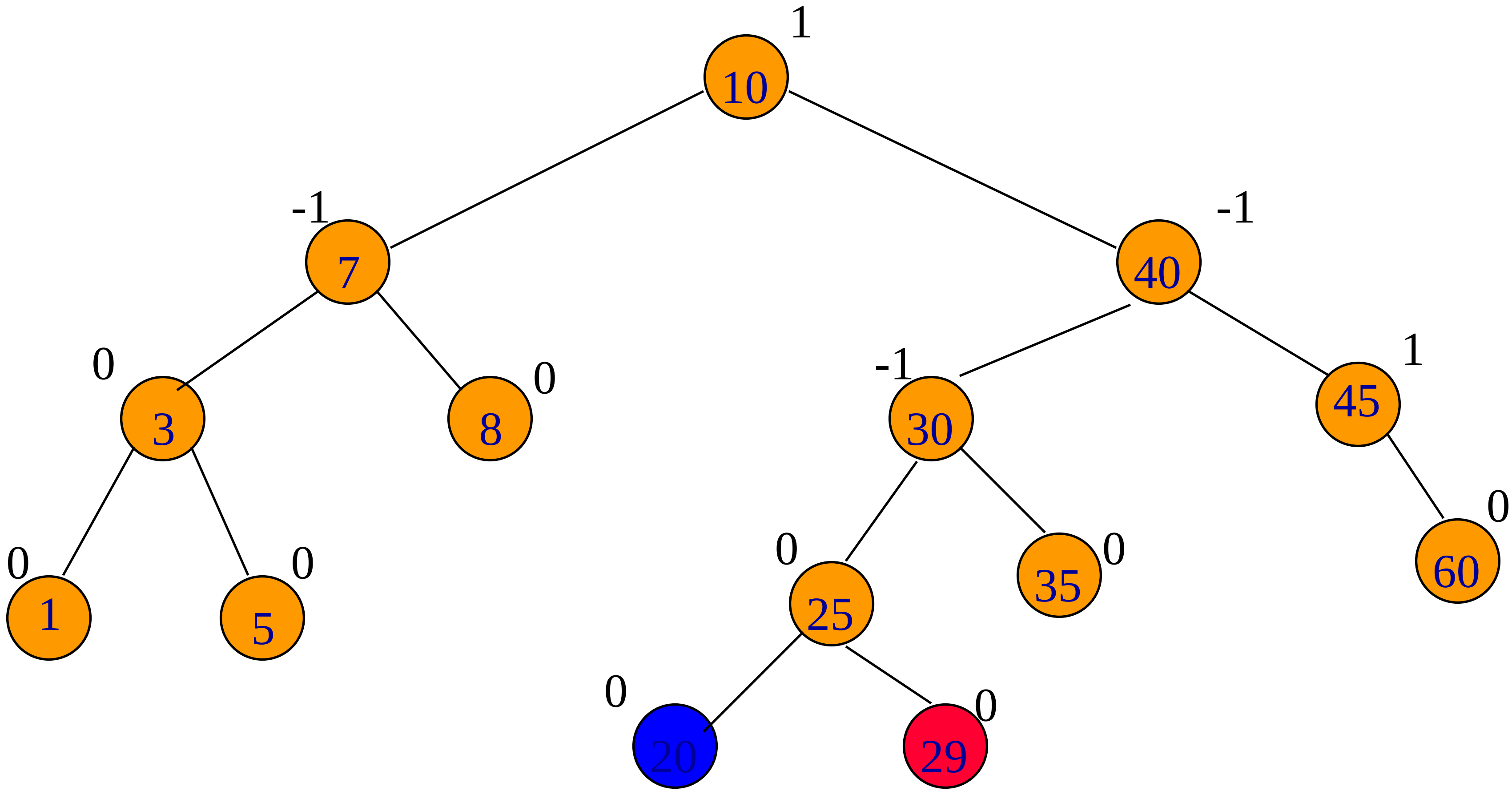


put(29)



RR imbalance => new node is in right subtree of right subtree of blue node (node with bf = 2)

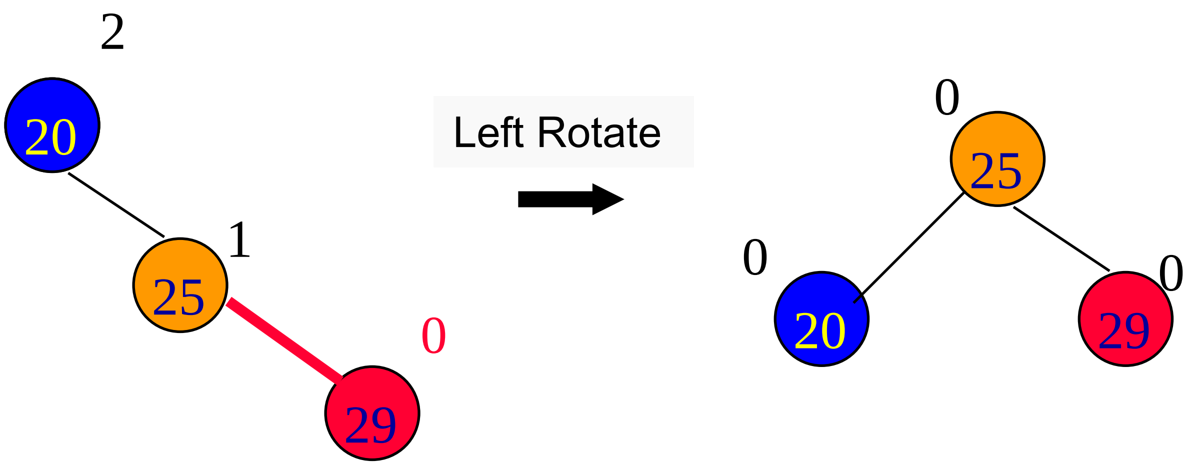
put(29)



RR rotation.

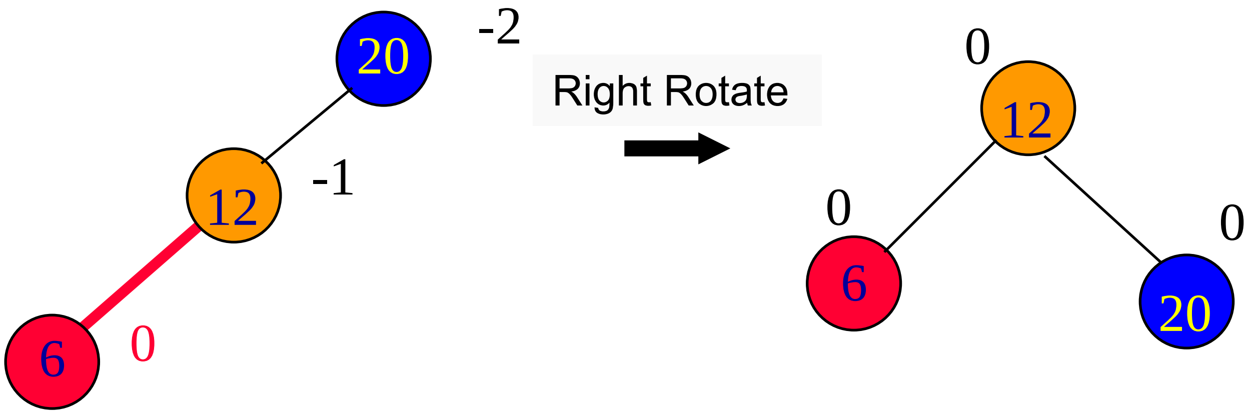
AVL Rotations

- RR



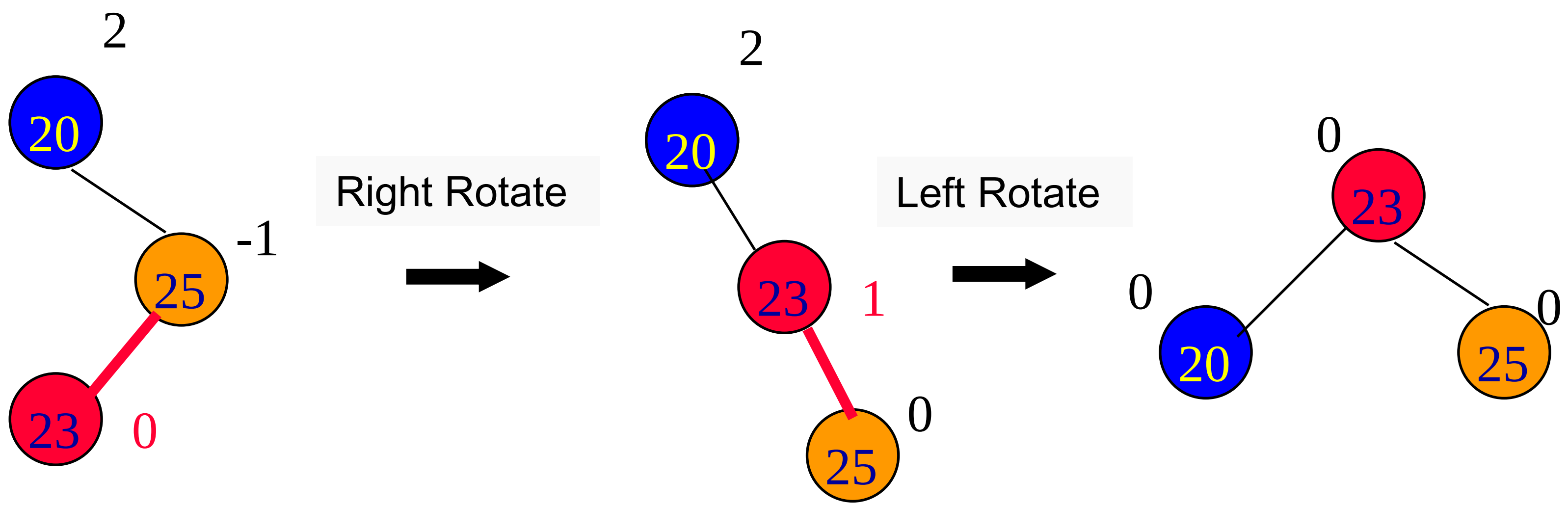
AVL Rotations

- LL



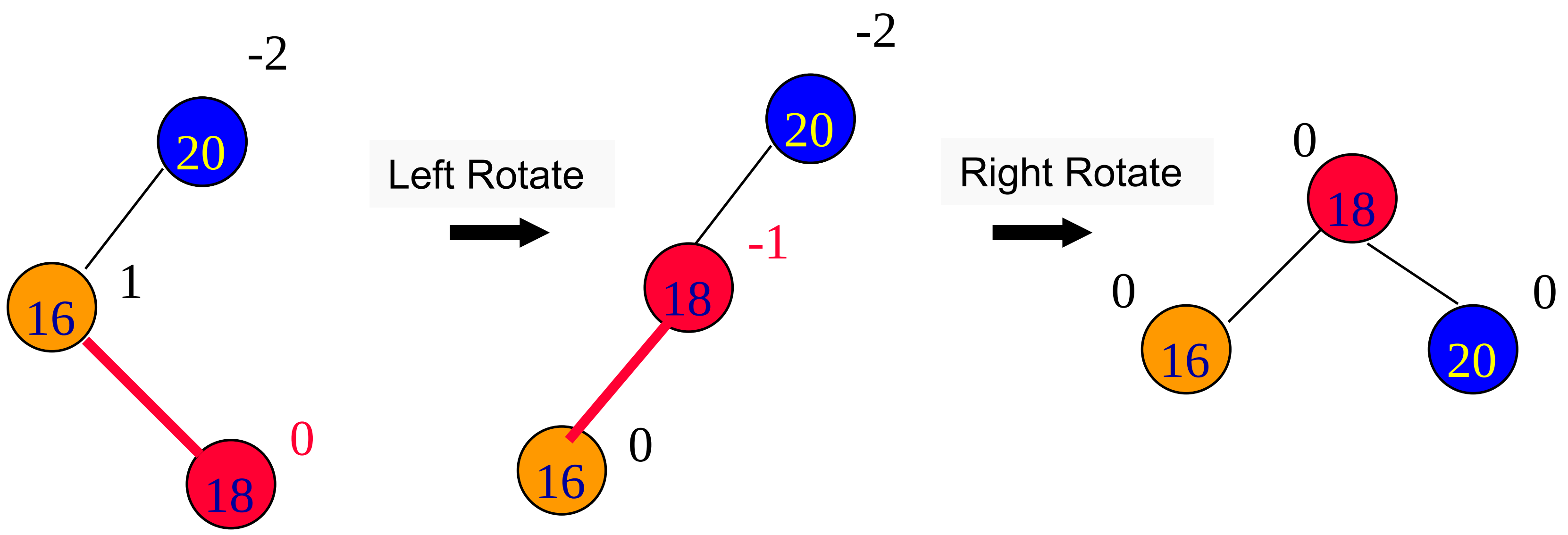
AVL Rotations

- RL



AVL Rotations

- LR

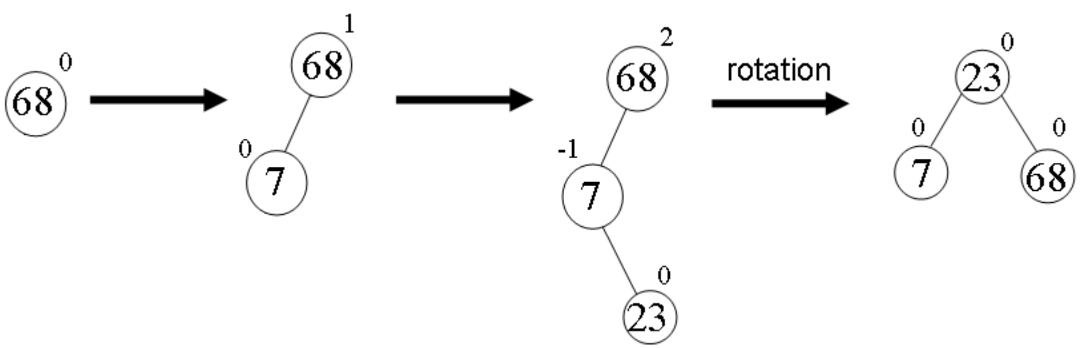


https://www.youtube.com/watch?v=yAFLlCZFJy0

Exercise

balance factor of node *i* = height of left subtree of node *i* -
height of right subtree of node *i*

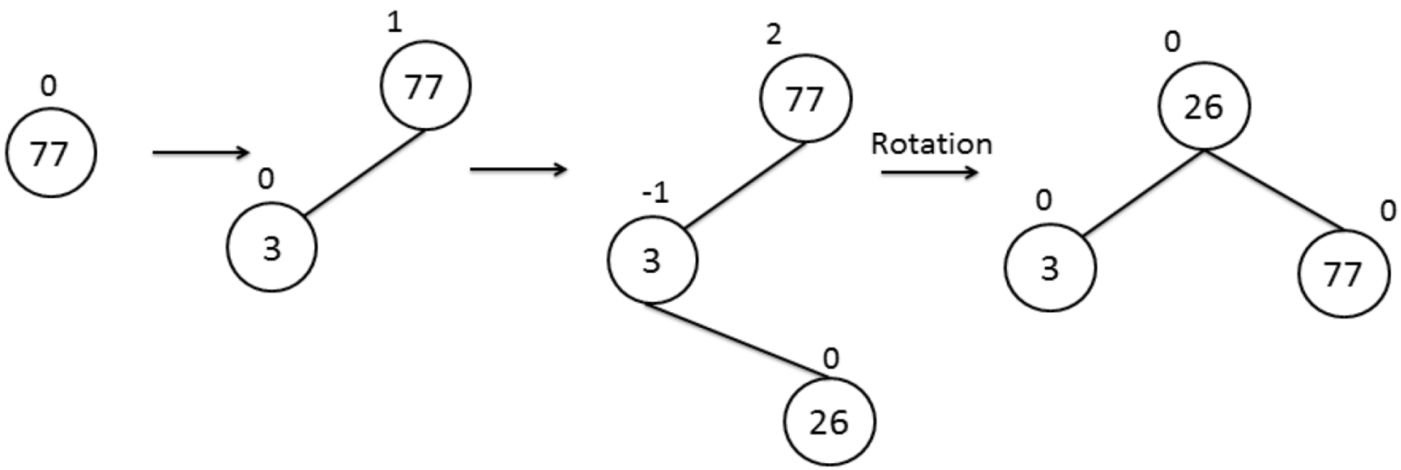
the sequence of integer keys 45, 3, 12, 52 is to be inserted into the AVL tree



Exercise

Suppose the sequence of integer keys 77, 3, 26 is to be inserted into an AVL tree based on the following balance factor for every node *i* of an AVL tree:

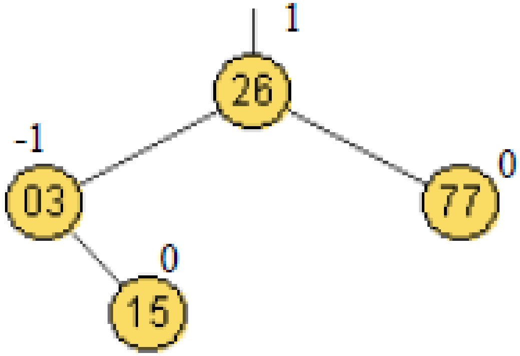
balance factor of node i = height of left subtree of node i - height of right subtree of node i



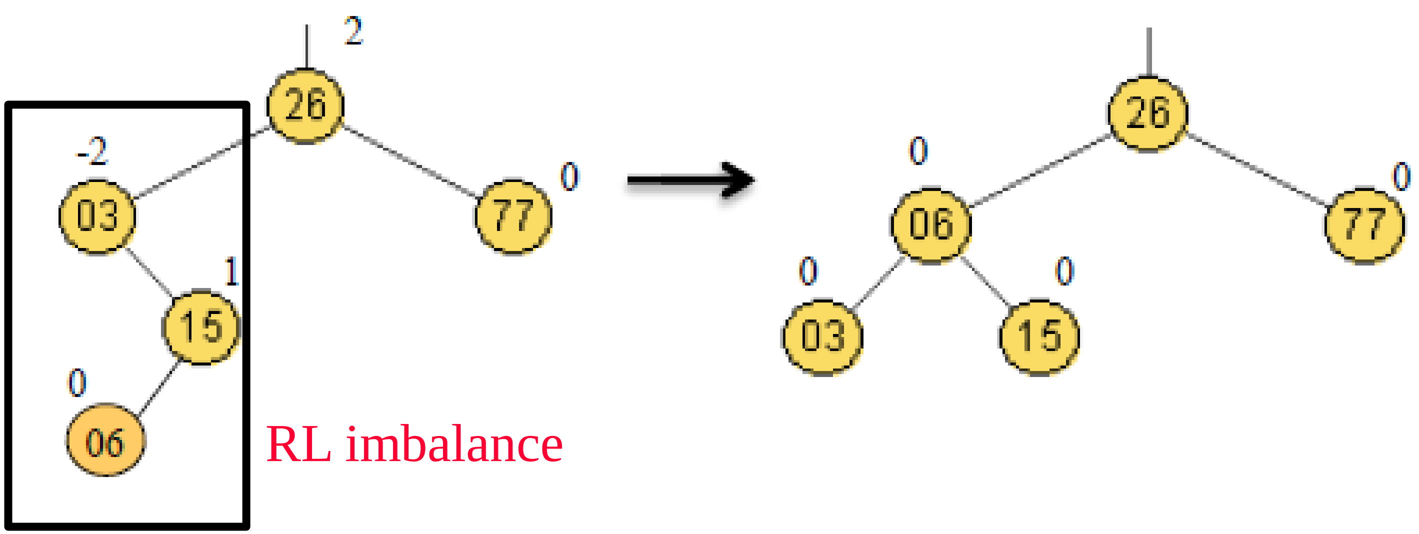
Above figure shows the keys with the balance factors of the AVL trees and necessary rotation for each insertion. Then, the sequence of integer keys 15, 6, 22 is to be inserted into the AVL tree with the keys 77, 3 and 26 in above figure. Draw the AVL tree, indicate the keys, balance factors and rotation if any after **EACH** insertion for the sequence of integer keys 15, 6, 22.

(10 marks)

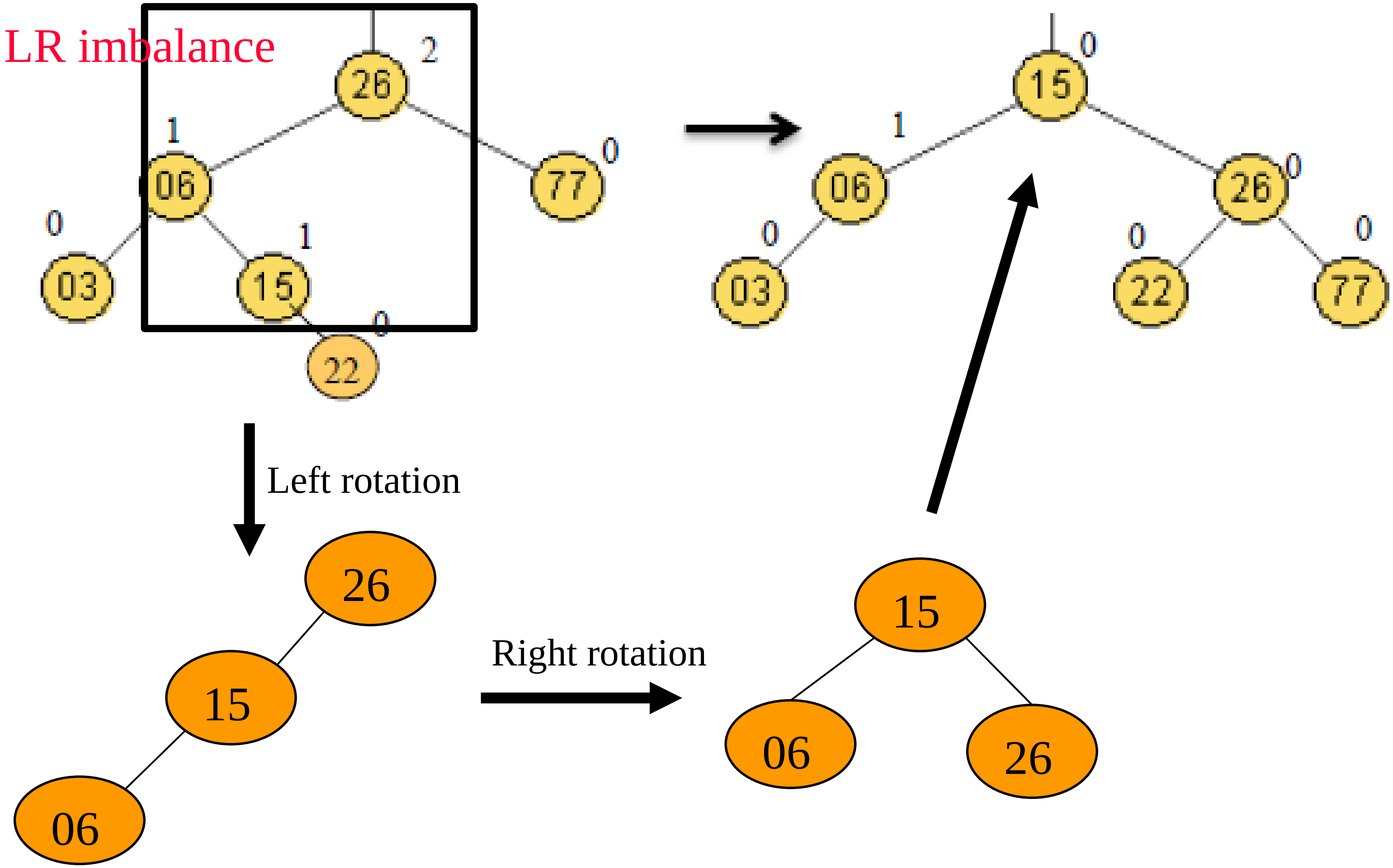
insert 15



insert 6 with rotation



insert 22 with rotation



Summary

- Binary Search Tree
- Operations
- AVL